Department of Computer Science

Technical Report

Undergraduate Dissertation: AnsProlog* Programming Environment (APE): Investigating Software Tools for Answer Set Programming Through the Implementation of an Integrated Development Environment

Adrian Sureshkumar

Technical Report 2006-07 **June 2006**

ISSN 1740-9497

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

AnsProlog* Programming Environment (APE):

Investigating Software Tools for Answer Set Programming Through the Implementation of an Integrated Development Environment

Adrian Sureshkumar

BSc (Hons) in Computer Science

2006

AnsProlog* Programming Environment (APE):

Investigating Software Tools for Answer Set Programming Through the Implementation of an Integrated Development Environment

submitted by Adrian Sureshkumar

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Batchelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed .....

This thesis may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed .....

Abstract

It has been recognised that better programming tools are required to support the logic programming paradigm of Answer Set Programming (ASP). In order to meet this demand, the aspects of programming in ASP that require better support need to be investigated, and suitable tools to support them identified and implemented. In this dissertation, an exploratory development approach is adopted to implement an Integrated Development Environment (IDE) for ASP: The AnsProlog* Programming Environment (APE). This is implemented as a plug-in for the Eclipse platform. Given that an IDE is itself composed of a set of programming tools, this approach is used to identify a set of tool requirements for ASP, together with some improvements to existing tools and programming practices.

Acknowledgements

I would like to thank my project supervisor, Dr Marina De Vos, for her continual support and guidance throughout the course of this dissertation.

I would also like to thank Martin Brain for giving up his time to discuss the project idea with me at the start of the year, participate in the demonstration and evaluation sessions and provide further support throughout the project. Many thanks also to Tom Crick for participating in the demonstration of the prototype system.

Finally, I would like to pass on my thanks to all who participated in the questionnaire.

Contents

List of Figures

Chapter 1

Introduction

Answer Set Programming (ASP) is a declarative programming paradigm with a semantics known as the answer set semantics (Baral, 2003). It is declarative in that the programmer specifies *what* needs to be achieved, rather than *how* it should it be achieved. It therefore lends itself naturally to applications in the domain of artificial intelligence, such as plan generation and reasoning in agents.

ASP programs, which are written in the language of AnsProlog*, are composed of a set of facts together with a set of rules from which other facts can be derived. A set of consistent facts that can be derived from a program using the rules is known as an answer set of the program. The possible answer sets for an AnsProlog* input program are computed with a program called a solver. Current solvers include the SMODELS and DLV systems.

A report by the Working group on Answer Set Programming (WASP)[1] acknowledges that better tools are required to support programming in this paradigm (Niemelä, 2005). However in order to identify the aspects that require better support, and consequently develop the appropriate tools to support them, a better understanding of the programming process is needed. Indeed, as argued by Seeley (2003), the improvement of these programming practices may be better at increasing productivity than the creation of new tools.

Nevertheless, the widespread use of programming tools in other paradigms is an indication of their value to the programmer. It is therefore important to investigate whether these tools could be applied to the domain of ASP and whether they would have the same impact as in other domains, in addition to identifying new tools to solve problems specific to ASP and improving programming practices. Thus the aim of this dissertation was to perform a preliminary investigation into these issues through the development of one such tool, the Integrated Development Environment (IDE).

The IDE is a tool that typically integrates the editing of source code with the tools to compile, execute and debug it into a single environment. Benefits of this approach can include:

- a faster development cycle, as code can be compiled automatically upon saving rather than by manually entering commands at a prompt.

[1] http://wasp.unime.it/

- a reduced strain on the mental resources of the programmer, through auto-completion mechanisms to prompt the user when they cannot recall what they want to type, and graphical user interface components to invoke commands.

For this dissertation it was chosen to develop an IDE, as although there are well known implementations for object-oriented languages such as C++ and Java, this tool was not known to exist for ASP. Indeed the only similar system that was found was a graphical user interface for the SMODELS solver[2]. Furthermore, IDEs have been developed for other logic programming paradigms, such as Prolog, indicating the potential for this to be applied to ASP in a similar way.

In order to be able to design and implement a system, its *requirements* need to be defined. In other words, the services that it has to provide as well as any constraints that it has to adhere to (Sommerville, 2001). Given that an IDE is not a single tool, but many tools integrated into a single environment, any conceivable programming tool was potentially a requirement of the system. Therefore the investigation into programming tools for ASP would form an essential part of its requirements elicitation phase. The development of the IDE is thus a suitable context in which to perform this investigation.

Investigating potential programming tools is an open-ended task, and therefore could never be considered to be completed, especially not within the timeframe of an undergraduate dissertation. It would therefore not have been appropriate to use the traditional waterfall model of software engineering to develop the system. In this approach, each stage of the software engineering process is completed before progressing to the next phase (Sommerville, 2001). As no 'complete' requirements specification would ever be produced, no design or development could ever occur, rendering any work performed useless.

Therefore in order to achieve an understanding of the requirements, whilst delivering a working system, an evolutionary development approach known as exploratory development was adopted. According to Sommerville (2001), the objective of this approach is to explore the requirements of the customer *and* deliver a working system. This fits naturally with the objectives of this dissertation.

1.1 Dissertation Outline

The structure of this dissertation will therefore follow this iterative process, rather than the traditional format of requirements, design, implementation and testing. However, for ease of reference a system requirements specification has been included in Appendix A, collating the requirements that are discussed in the course of this dissertation.

We first present a survey of some literature relevant to the dissertation, in order to identify some techniques that could be used and gain a better understanding of the domains of ASP and programming tools. We then consider how the initial requirements of the system were elicited through a brainstorming session and questionnaire, before examining why implementing the system as a plug-in for Eclipse met these requirements and was suited to the exploratory development approach.

[2] http://www.baral.us/bookone/ansprolog/

Following this, we discuss how proof of concept and prototype systems were developed and demonstrated to the users in order to further develop the requirements specification. Given this more detailed specification, we then discuss how some of the features that were identified were implemented in a series of increments, with observation sessions conducted after each release to continue the requirements elicitation process.

Finally, we present an overview of the testing strategy that was adopted and relate this to the exploratory development process, before considering the conclusions of the dissertation.

Chapter 2

Literature Survey

Before considering the process of requirements elicitation for the IDE in more detail, let us first consider some of the related background literature. In addition to discussing techniques that could be used during the course of this dissertation, this literature survey aims to introduce some important concepts from the domains of both Answer Set Programming and software tools.

Indeed, one of the key activities of the requirements engineering process is to develop an understanding of the domain of the proposed application (Sommerville, 2001). If the domain is not well understood, then important requirements may be overlooked or misinterpreted by the analyst, or even later by the system developer. Thus this literature survey is itself an important phase of the requirements elicitation process.

Let us begin our literature survey, with a consideration of the domain of Answer Set Programming and how this is situated in the wider field of logic programming.

2.1 Logic Programming

Logic programming stems from research into logic and automated theorem proving (Apt, 2003). In particular, the language Prolog (*Pro*grammation en *Log*ique) was created in 1973 by Colmerauer and Roussel (1993) and colleagues during a project to process natural language using automated theorem proving (Apt, 2003).

Two key features of this programming paradigm are that it is declarative and interactive (Apt, 2003). A declarative language is one in which the problem that needs to be solved is specified, rather than the algorithm used to solve it (Baral, 2003). Genesereth and Ginsberg (1985) present two advantages to this approach. Firstly that the program can easily be augmented with new information as required with no need for additional algorithm development, and secondly that the program can produce not only a result, but an explanation as to how it arrived at the result.

Apt also observes that logic programming has a procedural interpretation in addition to the declarative one. This is the method of computing the program solution, or in other words the algorithm used by the program interpreter. This process is interactive as once the interpreter has been started with a single program specification, the user can submit queries to it and receive answers.

In order that logic programming languages can model reasoning on incomplete knowledge, they cannot simply use classical logic (Baral, 2003). This is because classical logic is monotonic in that "*the conclusion entailed by a body of knowledge stubbornly remains valid no matter what additional knowledge is added*". Non-monotonic logic allows the addition of new knowledge to alter the validity of previously drawn conclusions, and thus better model human reasoning. Therefore, logic programming languages such as Prolog and AnsProlog* that need to model this kind of reasoning are non-monotonic.

According to Genesereth and Ginsberg (1985), most logic programming languages use a variant of the predicate calculus known as clausal form because this provides declarative semantics. Clauses, or rules, are sentences of the form $Head \leftarrow Body$, which can be read as $Body$ implies $Head$ (Robinson, 1992). In general, these are composed of several basic elements (Baral, 2003):

- Terms - variables, constants and n-ary functions applied to terms $f(t_1, ..., t_n)$
- Atoms - n-ary predicates applied to terms $p(t_1, ..., t_n)$
- Literals - atoms or their negations

The head of a rule, or its conclusions, is composed of a disjunction of literals, whereas the body, or its conditions, consists of a conjunction of literals (Robinson, 1992).

We can also define some special types of rule:

- Horn clause - one in which there is at most one conclusion (Robinson, 1992)
- Fact - a clause with a single atom in the head and no body (Baral, 2003)
- Constraint - a clause with no head (Baral, 2003)

2.2 Answer Set Programming

Answer Set Programming (ASP) is a branch of logic programming with a specific semantics known as the answer set (or stable model) semantics (Baral, 2003). According to Gelfond and Lifschitz (1988), the declarative semantics of each class of logic programming language are fixed in order to determine whether an answer to a given logic program is correct. This is achieved by specifying one of the models for each of the programs in the class as the canonical model. Syrjänen (n.d.a) provides an informal description of a stable model:

> *Informally, a model is stable if every atom in it has some "reason" to be there: for each atom in the model there has to be some rule that has the atom as a head such that the rule body is true in the model.*

In order that we can consider stable models (answer sets) in more detail, we first present an overview of other concepts in ASP as described by Baral (2003).

The language for ASP is known as AnsProlog* (*Pro*gramming in *Log*ic with *Ans*wer sets) or A-Prolog. The * in AnsProlog* is used to signify that no restrictions have been placed on the rules that can be represented by the language. However, there are several sub classes of the language that place such restrictions, such as

AnsProlog^{-not} for horn clauses. Some solvers, such as SMODELS and DLV, even extend some of the language features of these subclasses.

Baral defines an AnsProlog* program (i.e. with no restrictions) to be composed of rules of the following form, where each L_i is a literal:

$$L_0 \; or \ldots or \; L_k \leftarrow L_{k+1}, \ldots, L_m, \mathbf{not}\; L_{m+1}, \ldots, \mathbf{not}\; L_n$$

The answer sets of a program are "*defined in terms of the answer sets of the ground program*". Therefore let us now consider what is meant by the grounding of such a program.

A ground term is a term that contains no variables, and an atom (and consequently a literal) is defined as being ground if it only contains ground terms. The sets of all ground terms and atoms that can be formed from the constants and functions in a program, are defined as its Herbrand Universe and Herbrand Base. A rule is ground if it only contains ground literals, from which the grounding of a rule is defined as the set of all possible rules that can be generated by substituting ground terms from the Herbrand Universe into any variables in the rule. The grounding of the program is thus defined as the union of the grounding of all rules in that program.

The answer sets of a program are subsets of atoms from the Herbrand Base, known as Herbrand interpretations, that satisfy certain conditions (Baral, 2003). An interpretation of a program is an assignment of truth values to the ground atoms in the Herbrand Base, known as a valuation, such that any atoms in the interpretation are true otherwise they are false (Syrjänen, n.d.a). A rule in the program can then be said to be satisfied by the interpretation if it meets either of the following conditions (Baral, 2003):

- if all of the positive literals in the body are in the interpretation, and none of the negated literals in the body are in the interpretation, then the literal in the head is in the interpretation
- if there is no literal in the head of a rule (i.e. a constraint) then either the set of all positive literals in the body is not in the interpretation, or there exists a negated literal in the body that is in the interpretation

If a Herbrand interpretation satisfies all of the rules in a program, then it is called a Herbrand model of the program. For a program without negation as failure (AnsProlog^{-not}) its unique answer set is its minimal Herbrand model, where minimal means that no subset of this model is itself a Herbrand model.

However, AnsProlog programs that do allow negation as failure can have multiple minimal models of which some may not be intuitive. The example of this presented by Baral is the program, $a \leftarrow \mathbf{not}\; b$, which has the two minimal models $\{a\}$ and $\{b\}$, but there is no reason why b should be true.

The way in which answer sets of an AnsProlog program are defined is as follows. Given a potential answer set S, the program is transformed into an AnsProlog^{-not} program using the Gelfond-Lifschitz transformation. This deletes any rules whose body contains a negated literal from S, and any negated literals from the other rules. If the answer set of this reduced program is S, then S is an answer set of the original program.

Baral (2003) identifies several applications of ASP including:

- expression of database query languages
- planning and assimilation of observations in agents
- representation of constraint satisfaction problems
- solving combinatorial auctions
- applications which require reasoning on incomplete information

2.3 Programming Tools

Reiss (1996) defines a programming tool as "*any system that assists the programmer with some aspect of programming*". He identifies the following examples of such tools:

- Program Editors
- Compilers
- Linkers and Loaders
- Pre-processors
- Cross Referencers
- Source-Level Debuggers
- Debugging Aids

This definition can be expanded to include tools that support the wider software engineering process - this group is known as software tools.

Bruckhaus et al. (1996) observe that by supporting development activities that are not usually supported by tools, the introduction of software tools can help increase productivity. In addition they can also improve development processes by encouraging other activities, such as testing, to be undertaken. However, they observe that they can also have a negative effect on productivity by introducing new activities to perform or requiring an increased effort on certain existing activities.

Bruckhaus et al. also note how software tools can have different effects at different points in the software development process; "*a tool that is used in the upstream part of the development process may locally decrease productivity but help improve product quality or increase productivity downstream*". Although their study focuses on requirements management tools, their findings can clearly be applied to other software tools. An example of this would be the adoption of a unit testing framework, which could help reduce the cost of maintenance of a piece of software at the expense of a decrease in productivity during initial program development, given the overhead of developing the unit tests in parallel with the functional code. They stress that it is therefore important for a tool to be selected, whose impact will not have an adverse effect on the project for which it is being used.

2.3.1 ASP Tools

There are a limited number of tools available for ASP at the present. A report by the Working Group on Answer Set Programming (WASP) states that "*the need for better programming tools such as debuggers and tracers has been widely recognized*"

(Niemelä, 2005). However, it also mentions that progress is being made in the fields of debugging and equivalence testing. This project aims to help fill the gap in programming tools through the development of an IDE for ASP. However before considering IDEs themselves, let us examine some of the ASP tools that are available at present.

Smodels

The SMODELS system is a solver for answer sets of AnsProlog$^{\perp}$ and AnsProlog$^{\perp,\neg}$ programs with some extensions, together with primitive support for AnsProlog^{or} (Baral, 2003). The system consists of two command line tools, LPARSE and SMODELS (Syrjänen, n.d.a). LPARSE is a front end for SMODELS which performs a grounding of the input program and transforms it into the SMODELS input format (Baral, 2003). Syrjänen (n.d.a) identifies several alternative front ends to SMODELS:

- SMODELS API - a C++ library interface for calling SMODELS
- PARSE - SMODELS 1.x parser
- PPARSE - SMODELS 2.x parser for ground programs only
- MCSMODELS - Deadlock and reachability checker
- DLSMODELS - Deadlock checker

SMODELS, itself, is the engine that performs the computation of the answer sets of the input program (Baral, 2003). According to Syrjänen and Niemelä (2001), SMODELS has been used in application areas including planning, logical cryptanalysis and computation of stable models for disjunctive programs.

DLV

The DLV system supports disjunctive logic programming (DLP) under the answer set semantics, where a DLP program may contain disjunction in the head of rules and negation in the body (Leone et al., 2004). Baral (2003) defines this to be the computation of the AnsProlog$^{\perp,or}$ and AnsProlog$^{\perp,or,\neg}$ classes of AnsProlog* programs. He identifies the main difference between the DLV and SMODELS solvers, to be that SMODELS "*only has primitive functionality with respect to disjunction*" whereas DLV is centred on it.

noMoRe

The NOMORE (non-monotonic reasoning) system computes the answer sets of logic programs by computing non-standard colourings, known as a-colourings, of their associated block graph (Anger et al., 2002). These graphs model the dependencies between the rules, heads and bodies of a program (Linke et al., n.d.) and each answer set corresponds to exactly one a-colouring of the graph (Anger et al., 2002). Further detail on this technique is discussed in the work of Konczak et al. (2003).

NOMORE deals with variables by using LPARSE or DLV to perform a grounding of an input program containing variables (Linke et al., n.d.) before parsing the program

and computing its block graph (Anger et al., 2002). The graph itself can be visualised through the uDraw(Graph) tool[1] (formerly known as DaVinci), facilitating the structural analysis of programs. A visualisation of the graph can be completed at each step in the colouring computation process to demonstrate how the answer set is computed (Linke et al., n.d.). The system itself is implemented in Prolog and requires either SWI-Prolog[2] or the ECLiPSe Constraint Logic Programming System[3] in order to run (Anger et al., 2002).

IDEAS

In order to facilitate the computation of answer sets from similar sets of rules, the Interactive Development Evaluation tool for Answer Set programs (IDEAS) presented by Brain and De Vos (2004) provides an incremental computation approach. Rather than recomputing the entire answer set each time the program changes, as this is time consuming, the system makes use of previously computed results. It therefore lends itself readily to applications involving large, dynamic knowledge bases. However, Brain and De Vos do note that the algorithm used does not always produce an efficiency gain; all the answer sets may need to be recomputed in the worst case. It should be noted that this is also the first parallel answer set solver.

Brain and De Vos (2005) have since integrated debugging functionality into this tool. They describe the difficulties faced in determining whether an ASP program contains defects, as the only available indicators are the answer sets produced. This is in contrast to procedural programming, where exceptions and crashes clearly demonstrate the presence of these. They therefore classify this process as "*a task of supporting a programmer in investigating why a program does not behave as expected, rather than a series of static tests that can be performed*". This support is provided through query-based debugging, in which the programmer is able to determine the reasons for differences between the anticipated and actual answer set computed. These queries are of the form "*Why is set S contained in answer set A*" or "*Why is set S not contained in any answer set*". They acknowledge that other work on debugging has been performed by Satoh (2000) and Syrjänen (n.d.b) through powerful tools to "*analyse and 'repair' inconsistent programs*". However, they feel that "*they do not solve the complete problem of debugging*".

2.4 Integrated Development Environments

An Integrated Development Environment (IDE) is described by Delisle et al. (1984) as "*a set of tools that support program creation, modification, execution and debugging*". Boekhoudt (2003) observes that the initial constitution "*of editor, compiler and debugger*" has now been greatly expanded to include many other tools. Delisle et al. emphasise that the difference between the environment and its constituent tools is the way in which they are integrated within an environment - typically through the presentation of a consistent user interface and a common intermediate representation of the program being developed.

[1] http://www.informatik.uni-bremen.de/uDrawGraph/en/index.html
[2] http://www.swi-prolog.org/
[3] http://eclipse.crosscoreop.com/eclipse/

2.4.1 Integration Approaches

Reiss (1996) outlines three different approaches for integrating tools. The first, data integration, involves the sharing of information between the different tools through an intermediate representation of the program that is being developed. He observes that this has the advantage of allowing all the tools to directly access this representation, rather than each having to individually parse the code. This clearly has advantages with regards to performance and facilitating collaboration between individual tools, but as Reiss observes, it does have the disadvantage of being difficult to integrate new tools into the environment. This would clearly inhibit the integration of proprietary and closed source tools into such an environment, as the ability to modify such tools to use the internal representation would be limited without the consent and support of the tool developer.

However, another approach suggested by Reiss facilitates this at the expense of collaboration between the tools and a greater awareness by the user of the underlying tools that are being used. This approach provides a common front-end for several tools, usually in the form of a text editor, such as emacs. This permits tools with a textual interface to be invoked and their output displayed from within the editor, providing the impression of an integrated tool. We discuss some features of emacs in more detail later.

The final approach he presents is message passing between the individual tools, more commonly via a message server than "*point-to-point*". This clearly facilitates the communication between the individual tools, but, as he remarks, it does not allow the sharing of large amounts of data such as a program's abstract syntax tree.

It is clear that a combination of these approaches would enable the environment to achieve a balance between the number of tools integrated and the tightness of their integration. Reiss notes that this is the approach used by actual environments.

2.4.2 Benefits and Limitations

Weinberg (1998) describes some of the consequences of psychological set on programming. This phenomenon may cause a subject to perceive something incorrectly, simply because they are expecting to perceive it in a particular way. Weinberg cites the example of skim reading a text and ignoring misspelt words because the subject is expecting to see the correct word. This is clearly an issue in programming, given that the compiler will interpret the code as actually written, whereas the programmer may interpret it as what he had intended to write. This can be compounded by other issues, such as the font that is used. For example, it can be difficult to distinguish between the number 1 and the letter l in the Courier font - a font commonly used in editors.

In order to assist the programmer in recognising these mistakes, Weinberg proposes a list of techniques. These include setting off keywords in bold and stripping comments from the code listing. The former will make it more evident to the programmer that they misspelt a keyword, or flag, for example, that they have used a keyword for a variable name. Extending this technique to other syntactic subclasses and augmenting it with colour provides the commonly used feature of syntax highlighting. The latter technique of being able to strip out comments, helps to eliminate the effect of

comments explaining how a section of code works from convincing the reader that the code does indeed work as suggested.

Features of IDEs and editors such as syntax highlighting, prettyprinting (indenting), bracket matching and underlining of errors, all aim to make programs easier to read and therefore easier to spot errors. However, as observed by Sheil (1981) a given formatting scheme may not be suitable for all users, as a user familiar with another scheme may find it difficult to adjust to the new layout. It is therefore important that an editor should provide the user with the facility to configure the formatting scheme to that with which they are most familiar.

As observed by Curtis (1982) human memory can be divided into long term and short term memory:

> *Short term memory is a limited capacity workspace that holds and processes those items under attention. Long term memory is a store which retains knowledge over long time periods without our being conscious of it.*

Given that the capacity of short term memory is limited to approximately 7 items, the provision of tools to relieve the strain on the programmer's memory would clearly be welcomed. Fry (1997) presents his 'Emacs Menus' system as an example of a programming environment that uses the computer's memory to overcome the limitations of biological memory. As the programmer inputs a program, the system dynamically calculates the possible choices and proposes these to the programmer via a pop-up menu. This enables the programmer to concentrate on completing their task, rather than trying to determine what to type.

In addition to any productivity gain from this, the insertion of code by the environment would eliminate simple spelling errors that may not be caught until compile-time. Fry identifies other benefits of this system in relation to reducing errors, such as enabling the programmer to supply the correct number and types of arguments to functions. Indeed, code completion is a technique found in many IDEs today. Although the syntax of logic programs is limited to clauses, there is still clearly some scope for this technique. For example, the predicates defined in a logic program could be parsed as the program is written, and used to provide automatic completion of predicates as the user types.

There are clearly other benefits to IDEs than those we have considered, such as the encapsulation of command line tasks into menu options and button presses. However it is beyond the scope of this literature review to perform a detailed discussion of all of these. Instead, let us now consider some of the limitations of these tools.

It is argued by Seeley (2003) that "*a new IDE that reduces compile/edit/debug turnaround is optimizing the wrong problem*". Instead, he proposes that it would be better to reduce the number of defects in a piece of software, before reaching the debugging process, by using better working practices. Although it is clear that these will help improve productivity and that programmers should endeavour to adopt these, it does not mean that the use of an IDE as a tool to increase productivity should be devalued. You could not say that a programmer who adopts the 'best' working practices known would not benefit from a faster compile/edit/debug cycle, nor those who have yet to adopt better working practices or are in the process of doing so.

Perhaps a more rational criticism of IDEs suggested by Boekhoudt (2003) is the tendency of IDE vendors to try and include as many features as possible, which he describes rather succinctly as confusing "*integration with housing it all under one roof*". He quotes the high cpu, memory and storage requirements of many current IDEs, such as Sun One Studio and Visual Studio.NET, as well as the shortfall in some of the bundled tools, for example XML and HTML editors. This forces the user to install more comprehensive tools on their machines and switch between the IDE and these additional tools, further compounding the strain on system resources and increasing the time taken to develop software.

Seeley (2003) also remarks that as many people only use a subset of a complex tool's features, the effectiveness of the tool is limited to the effectiveness of the subset that is actively used. It is clear from this that in order to avoid these concerns IDE developers need to concentrate on developing the user's core feature set, and make it easier for the user to use their preferred tools from within the IDE

2.4.3 Extendable Frameworks

Certain IDEs provide interfaces for developers to integrate their tools into the environment as 'plug-ins' (Boekhoudt, 2003). This approach can enable the user to select the most suitable tools for their needs from within the environment, rather than having the choice of either working between multiple applications or using a poor alternative tool from within the environment. He also observes that this modular approach gives the developer the opportunity to remove any features that they do not require, saving both start-up time and system resources.

In addition, these frameworks clearly reduce the development cost of an IDE by removing the need to implement the structure for the user interface and providing common features such as version control. This clearly makes them well suited to scenarios like this project, where the aim of evaluating the suitability of various language features needs to be completed within a relatively short time-frame. This could clearly not be achieved if the basic infrastructure would have to be implemented from scratch, as this would limit the time available to develop the features themselves. The main caveat of this approach is that the overall usability of an IDE based on one of these frameworks, would be limited by the usability of the chosen framework.

Examples of these frameworks include tools such as NetBeans, Eclipse and Emacs, of which we will examine Emacs and Eclipse in more detail.

Emacs

Curley (2002) describes how the Emacs editor can be viewed as an extendable IDE, through the development of a simple 'Hello World' application in C. In addition to editor features such as indentation and syntax highlighting, he demonstrates the possibility of performing tasks such as directory manipulation, compilation, debugging, version control and file diffing. The customisability of Emacs is facilitated through packages known as modes, which can be further customised by editing the .emacs file. The flexibility of the tool is further increased by its free and open-source nature.

This could therefore provide an initial starting point for the development of an IDE for ASP. Indeed, LPARSE/SMODELS are already supplied with a major mode for Emacs that offers indentation, syntax highlighting and running LPARSE and SMODELS via commands (Syrjänen, n.d.a). Regardless of whether the IDE would be based on this, it should be able to support the syntax highlighting and indentation styles provided by this mode in order to facilitate the transition of existing users to the IDE. However, one limitation of this system appears to be the installation method described in Syrjänen's user manual. It requires the user to edit their .emacs file, which may not be an appropriate task for a novice user.

Eclipse

The Eclipse platform, described as "*an IDE for anything, and for nothing in particular*" (Eclipse, 2003), is surrounded by an industry buzz according to Wolfe (2003). He identifies several reasons for this success including being free, given that equivalent IDEs can cost more than $1,000, and supporting multiple platforms including Windows, MacOS, Solaris and Linux (Red Hat & SuSE). The platform provides a lot of generic functionality and "*is built on a mechanism for discovering, integrating, and running modules called plug-ins*" (Eclipse, 2003). Moreover, the licence terms allow third-party developers to charge for any extensions that they produce (Wolfe, 2003), which clearly provides an incentive for developers of commercial and open-source tools to use this platform. It is however criticised by some for having an excess of features, which could be overwhelming for inexperienced users.

Plug-in's typically consist of Java code contained in a JAR (Java Archive) file, together with resources and a manifest file (Eclipse, 2003). The development of plug-ins is facilitated by the provision of an IDE in Eclipse - the Plug-in Development Environment (PDE). The manifest file is an XML file which defines a set of extension points, which other plug-ins may extend, together with its extensions - how it is extending the extension point defined by another plug-in. This could clearly be an advantage in an IDE for ASP, by allowing developers to integrate their own solvers into the framework provided. The best known plug-in for Eclipse is probably the Java Development Tooling (JDT) included in the main distribution together with the platform and PDE - although the platform is also available separately. Wolfe observes that this is probably why Eclipse is viewed by many as simply a Java IDE, rather than a framework to host IDEs and other tools.

2.4.4 IDEs for Logic Programming

Komorowski and Omori (1985) and Francez et al. (1985) describe the situation in the 1980's, in which little progress had been made with respect to programming environments for logic programming. Indeed they observe that the environments of the time were restricted to imperative and functional languages. This is clearly no longer the case, given that a quick Google search for *Prolog IDE* generates many pages of results for tools such as Visual Prolog[4] and the Prolog Development Kit[5]. There even exist environments written as plug-ins for Eclipse, such as the Prolog

[4] http://www.visual-prolog.com/default.htm
[5] http://www.bettersoft.ro/IDE-pdk-BetterSoft.php

IDE implemented as part of the JTransformer project[6].

However, performing similar searches for ASP does not return any relevant results. In fact, at the time of writing the top search result on Google for the query *"Answer Set Programming" Integrated Development Environment* was the idea for this project, demonstrating that this is indeed one of the areas of tools for ASP that is underdeveloped. Thus it can be said that we find ourselves in a similar situation today with ASP, as Komorowski and Omori, and Francez et al. did in the 1980's with logic programming in general.

The fact that several environments exist for the Prolog language indicate that the development of IDEs for logic programming languages can be achieved and warrants an investigation into whether this would also be possible for ASP.

2.5 Usability Evaluation Techniques

In order to investigate whether the various IDE features that are proposed in this dissertation do indeed provide greater support to the programmer, it is clearly important to be able to assess how ASP development undertaken with the IDE compares with development using solely the existing tools that are available. Although a detailed usability study was not performed in this dissertation, a consideration of various usability evaluation techniques and the way in which they can be applied to evaluating the IDE would be useful for considering this aspect in any further work.

The first question that needs to be answered is how the differences between using the IDE and the existing tools can actually be measured. According to Constantine and Lockwood (1999) this can be achieved with usability metrics - "*quantitative indices that measure or estimate some factors or dimensions of software quality*". However, they also note that they should not be used in isolation to other evaluation approaches and should be used in the right context in order to avoid the overvaluation of their significance. Some examples of these will therefore be considered as part of the discussion of the evaluation techniques to which they can be applied.

Moore and Redmond-Pyle (1995) identify the following three evaluation approaches: reviews, user testing and surveys. In addition to these, Shneiderman (1997) also considers acceptance tests, evaluation during active use and controlled psychologically oriented experiments, although we will not examine these as part of this literature review.

2.5.1 Reviews

Reviews entail the evaluation of the GUI by a Human Computer Interaction (HCI) expert or through the use of a set of heuristics (Moore and Redmond-Pyle, 1995). Although a HCI expert can quickly identify usability issues in a piece of software, Shneiderman (1997) notes that the quality of the evaluation may be affected if the experts do not have a good working knowledge of the domain in which the software is to be used. In addition, it may be difficult to apply this technique when evaluating the IDE for ASP, given that the human resources required may not be

[6] http://roots.iai.uni-bonn.de/research/jtransformer/prologide

readily available.

Evaluation using a set of heuristics (Nielsen and Molich, 1990) would be more suited to providing a quick evaluation of the IDE as it is much cheaper and therefore "*feasible in low-budget projects*" (Moore and Redmond-Pyle, 1995). However it is clear that the quality of the heuristics used and the knowledge of the person applying them will affect the quality of the evaluation.

2.5.2 User Surveys

User surveys can be employed through a structured interview or questionnaire (Moore and Redmond-Pyle, 1995). The main advantage that they give for questionnaires is the ability to produce quantitative results through their deployment to a large user group. Shneiderman (1997) presents online surveys as a way of reducing the costs associated with the paper-based approach. This clearly also facilitates the distribution of questionnaires to a wider geographical user group enabling a more representative sample of users views to be collated. Interviews can be useful when applied to selected respondents of a questionnaire in order to elicit more information on any points raised (Moore and Redmond-Pyle, 1995).

Questionnaires can be viewed as a type of preference metric, collecting subjective opinions of the user with regard to the system under evaluation (Constantine and Lockwood, 1999). As they observe, subjective opinion is only an indication of the user's preference and not necessarily of true usability. However, they also remark that it is potentially a better indicator of marketability, which is clearly an important factor that needs to be considered when attempting to gain a user group for a new piece of software. One standard set of preference metrics that Constantine and Lockwood advocate is the Software Usability Measurement Inventory (SUMI). This takes the form of a questionnaire consisting of 50 questions that is completed by a user subsequent to a trial of a working version of the system. Other approaches that exist include the Subjective Usability Scales for Software (SUSS) (Constantine and Lockwood, 1999), Questionnaire for User Interaction Satisfaction (QUIS), and the Post-Study System Usability Questionnaire (Shneiderman, 1997).

Another factor that needs to be considered when performing an evaluation via survey or usability testing is to select participants that are not involved in the design process as their views may be biased by their attachment to the system (Moore and Redmond-Pyle, 1995).

2.5.3 Usability Testing

Usability testing involves observing or monitoring users carrying out a specific set of tasks, and analysing the results to gain an indication of the usability of the system under test (Constantine and Lockwood, 1999). This method of testing has ethical concerns that need to be addressed. Shneiderman (1997) observes that it is important for any users to be informed about what they are expected to undertake and that it is the system and not them that is under test. He also makes it clear that participation should be voluntary and the consent of the participant should be sought, for example by signing a statement.

Simply applying this technique to determine how a usable a system is, is not likely

to generate useful results, rather the tests need to be targeted to answer specific questions (Constantine and Lockwood, 1999). One of the examples that they provide is that of comparing alternative designs for a particular problem. This could clearly be extended to a comparison between a new and existing system by evaluating whether the new system improves upon known areas of poor usability in the existing system. However this would need to be used in combination with other techniques to ensure that the evaluation does not ignore any new usability problems that may have been introduced by the new design.

This generally takes place in one of two settings; a usability laboratory or in the field (Shneiderman, 1997). As usability laboratories give the evaluators the ability to control experimental conditions such as physical surroundings and distraction levels, this approach should be suited to a comparison of two systems by ensuring that both systems can be tested in the same conditions. However, the main limitation is that the evaluation is taking place in an unnatural environment, a field test could therefore provide a better indication of the usability of the system at the expense of control over the test conditions (Constantine and Lockwood, 1999). A laboratory test to compare a new and existing system could be followed by a field test of the new system in order to gauge its usability in a real environment.

Constantine and Lockwood describe performance metrics as a way to "*quantify and summarize important aspects of actual usage either under controlled laboratory conditions or within an ordinary work environment*". They put forward a suite of six measures identified in the work of Bevan and Macleod (1994):

- Completeness
- Correctness
- Effectiveness
- Efficiency
- Proficiency
- Productiveness

These values can be compared with other data gathered during the evaluation in order to support or undermine the assessment of the usability of the system gained through this other data. For example, a contradiction between the two data sets for one particular user could be used as an indicator of an oversight when performing an observation of the user performing a task. If the session was recorded on video, the session could be reassessed to determine whether this was indeed the case.

Shneiderman (1997) identifies two limitations of usability evaluations. The first is that the evaluations generally consider a user's first time usage of the system. This needs to be considered when performing a comparison between a current method of working and a new approach. It is clear that users familiar with a well established way of working will perform a task more productively using this than with a new approach that they have to learn whilst attempting to perform the task. In order to remove this bias, it may be better to test users new to the domain that have no experience of a particular methodology. However, the value of this approach would be limited as the inexperienced users would have to spend time learning the domain, in addition to the methodology. If this were to be extended over a longer period of time, this could become an evaluation of the best approach for learning a language. The other factor that he considers is that during an evaluation session of between

two to fours hours, a limited coverage of the system's features is possible. Thus, it would clearly be better to extend the evaluation over a longer period.

2.6 Conclusion

The consideration of the paradigm of ASP at the start of this literature survey, has introduced some of the terminology used in this domain (such as rules, grounding and answer sets) prior to the requirements elicitation process. As this is the domain that the IDE intends to support, this should serve to provide a better understanding of the requirements expressed by the user. In addition to the concepts outlined here, key resources such as the book by Baral (2003) were identified during the literature search, serving as reference for any further terminology requiring clarification.

Given that the dissertation aims to investigate how programming tools could better support ASP, the examination of programming tools has illustrated the importance of choosing the correct tools in order that the programmer is not hindered rather than supported. Additionally, the subsequent of review of existing ASP tools identified the SMODELS, DLV, NOMORE and IDEAS solvers, providing a starting point for considering which tools need to be integrated into the IDE.

The discussion of IDEs has identified some of the human factors, such as psychological set and the limitations of short term memory, that can affect the programming process. Considering these factors and the constraints they place on the programmer would help to identify areas that could be supported by tools, and guide the design of tools to do so. Furthermore it has demonstrated some features present in other IDEs that help to support these aspects, such as syntax highlighting and autocomplete, allowing them to be considered for inclusion in this system.

Aside from the benefits provided by these features, some criticisms of IDEs were considered such as the tendency of IDEs to try and include as many features as possible. Having identified these aspects, they could be taken into consideration during the design of the system in order to avoid the same mistakes being duplicated in this dissertation. In addition, the potential for improved working practices to be of more benefit to the programmer than additional tools demonstrated that this investigation should also consider any improvements to these as alternatives to any tools and features proposed.

Guidance on the design and implementation of the IDE can also be drawn from the literature survey. For example the relative merits and deficiencies of the various integration approaches that we have discussed could be taken into account when attempting to integrate a tool into the environment. Furthermore the benefits, in terms of reduced development time and extendibility, of writing plug-ins to integrate tools into an existing environment have made this the favoured approach for implementing the IDE, as discussed further in chapter 4.

The consideration of IDEs for other logic paradigms, such as Prolog, has lent support to the feasibility of developing the tool for the similar paradigm of ASP. Moreover, an evaluation of the IDEs identified could help to elicit further requirements for this system, although this is not considered for this dissertation.

Finally, the importance of being able to compare the use of the IDE with the existing

ways of working was identified together with the usability evaluation techniques required in order to do this. Although this comparison is not performed in this dissertation, the discussion served to guide the choice of any evaluation techniques employed.

Chapter 3

Elicitation of Key Requirements and Potential Features

Given that an exploratory development approach has been used for this dissertation, this initial requirements gathering phase did not aim to produce a complete requirements specification for the IDE. It did, however, aim to ascertain some of the key constraints on the system in order that a suitable development approach for the exploratory system could be chosen. These constraints on a system are also known as its non-functional requirements (Sommerville, 2001).

Conversely, the functional requirements define the behaviour of the system and the services that it should provide. For this system, this includes the features that would need to be incorporated into the IDE. In order to begin the development of the exploratory system, an initial set of these features needed to be elicited. This was thus another aim of this phase of the requirements engineering process.

Given the time constraints of the dissertation, not all of the features that have been considered in this phase have been implemented. However, given that these are all potential requirements of the system, they should still be explored as part of the wider investigation into programming tools for ASP. The features that have been considered are indicated at the end of this chapter.

Let us now consider the process by which the key requirements and potential features were elicited and the reasons for using this approach, before considering some of the requirements and features that were obtained.

3.1 Elicitation Process

In order to develop a set of requirements for a system, the people who have a vested interest in the successful development of the system need to be identified - these are known as the stakeholders (Preece et al., 2002). The primary stakeholders for the IDE, are thus ASP programmers and other members of the ASP community, as they stand to benefit from the improved tool support that the IDE could provide.

Therefore in order to gather the fundamental requirements for the IDE and a list of potential features, a questionnaire was developed and distributed by e-mail to members of the ASP community. A copy of the questionnaire and a summary of the

results can be found in Appendix B.

This was deemed to be a suitable technique for obtaining an overview of the requirements of the community in general, as it provided a cost effective means of gathering data from users around the world rather than solely from those at the University (Preece et al., 2002). Alternative approaches such as face-to-face interviews, or observation would clearly be infeasible for an undergraduate project given the time and monetary cost of travelling. Even telephone interviews would have been difficult to co-ordinate and may have imposed a greater time burden on its participants than a questionnaire itself. Whilst the latter approaches are potentially richer sources of data, in the context of this project they would be better used after the questionnaire as follow-up interviews to help elaborate on any interesting points.

Although questionnaires can suffer from very low response rates, some less than 30% (Ruane, 2005), it was felt that the potential benefits that an IDE could provide would encourage participants to respond. Furthermore, the questionnaire was designed to be short and consist mainly of closed questions in order to minimise the time required by the participant to complete it, although a few open questions were included to allow further elaboration if required. Despite this, only 17 of the 48 questionnaires distributed were returned, giving a response rate of 35%. Although this may simply have been due to an unwillingness to complete the questionnaire, the possibility of a lack of interest in the IDE by those that did not respond should also be considered (all respondents indicated that they would be interested in using the IDE). However, it was felt that even if this were the case then it would still be valuable to continue to pursue the project, given that there were still some users that were interested in such a system.

The experience of the participants in ASP development ranged from 1 to 10 years experience, with 4 years experience on average. Only 4 participants of the 16 that responded to the question had less than 3 years experience. This suggests that the participants have sufficient knowledge about the process of developing in ASP to provide valuable feedback on how this could be better supported. However it is also possible that through their years of using the current ASP development tools, the participants may be less aware of areas that need better support, as they have learned to work around them. It could be useful to target a survey at more novice users, as these may be able to spot more easily what is lacking.

3.2 Key Requirements

3.2.1 ASP Tools

In addition to the development of new tools to support the programmer, an IDE also integrates existing tools, and as discussed in the literature survey (section 2.3.1) there are already various tools available for ASP that could be integrated into the system. Moreover, the solver is the tool that performs the computation of a program's answer sets and defines the syntax of the programs that would be developed within the IDE. It was therefore important to consider which solvers would need to be supported by the IDE as this choice would constrain the development of other tools and features.

The first question on the questionnaire aimed to determine which tools were used

by the participants. It proposed the tools that had already been identified during the literature survey, but provided space for respondents to include any other tools that they use. This had the aim of identifying any other tools that needed to be integrated into the environment, in addition to determining which solvers needed to be supported.

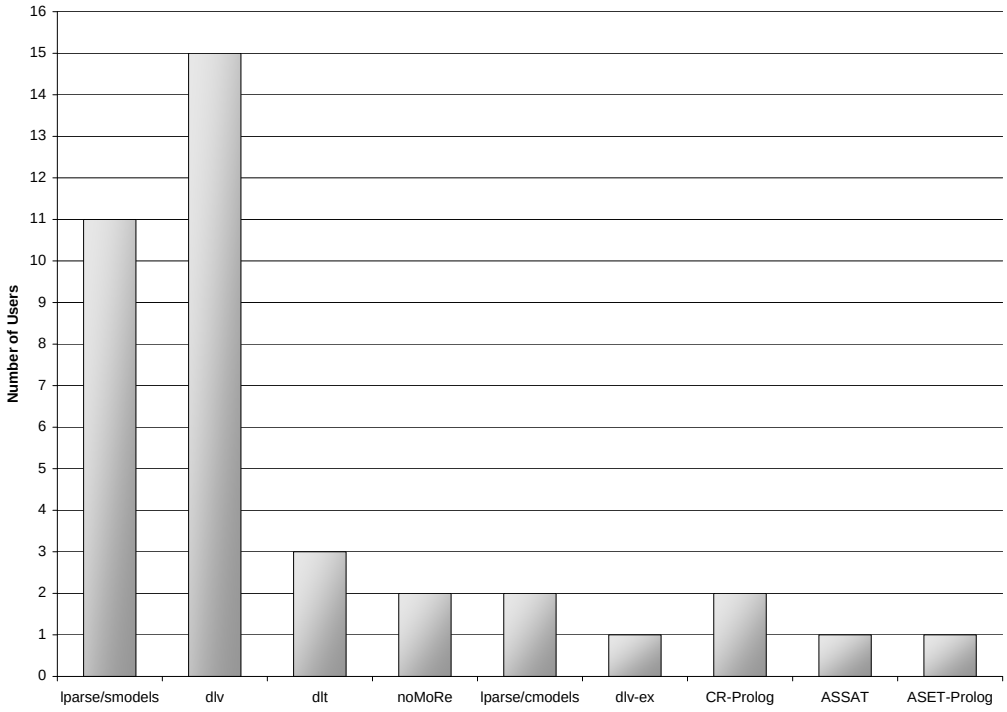


Figure 3.1: ASP Tools Used by Participants of Questionnaire

The results of the questionnaire showed DLV and LPARSE/SMODELS to be the ASP tools most widely used by the participants (Figure 3.1), although this was not surprising given that they are arguably the most well known solver implementations. It was clear from this that support for these two solvers would need to be included in an IDE.

However given the time constraints available, it was decided that for this dissertation only one of these solvers would be supported. This would allow more time to be spent exploring a wider range of supporting tools and IDE features, rather than applying a smaller subset of features to multiple solvers. Whilst each feature could have been applied to a different solver, the overhead of integrating each solver into the environment would still have had to be incurred. Furthermore, this approach would have led to an incoherent exploratory system, rather than something that would be of immediate benefit to its users.

Although it had a slightly lower response than the DLV solver, it was chosen to develop the IDE around the SMODELS solver and LPARSE front-end. Given that it is an open source product under the GNU General Public Licence (GPL)[1], whereas only binary builds are available for DLV, the possibility of code reuse was available.

[1] http://www.gnu.org/copyleft/gpl.html

This would potentially allow some features to be developed more quickly, allowing a greater number to be considered within the time constraints of the project. Moreover, as this solver is used by members of the department at the University, supporting it would facilitate the possibility of evaluating the system.

Nevertheless, given that a complete system would need to support DLV, the system produced as part of this dissertation would still need to facilitate the future integration of this solver.

Beside the tools that had been suggested to the respondents, the questionnaire also identified five other tools that had not previously been considered:

- CMODELS[2] - "*an answer set solver that uses SAT solvers as search engines*" (Lierler and Maratea, 2004).
- DLV-EX[3] - An implementation for the DLV system of "*Answer Set Programming with External Predicates (ASP-EX), a framework aimed at enabling ASP to deal with external sources of computation*" (Calimeri and Ianni, 2005).
- CR-MODELS[4] - The inference engine for CR-Prolog, "*an extension of A-Prolog by consistency restoring rules with preferences*" (Kolvekal, 2004).
- ASSAT[5] - A system that computes "*answer sets of a logic program by using SAT solvers*" (Lin and Zhao, 2004).
- ASET-SOLVER[6] - A solver for ASET-Prolog, "*an extension of A-Prolog that adds to the language sets of terms and functions from these terms to natural numbers*" (Heidt, 2001).

Given the range of tools used by members of the community, it would clearly not be viable to attempt to provide support for every one of these. Equally, it would clearly be impractical for users of these tools to develop a program from within the IDE, but solve it, say, from the command line. Consequently, this could limit the user base of the system. Given this, it was clear that the IDE would need to provide some sort of extension mechanism (e.g. plug-ins) or the ability to run external commands from within the IDE. This would allow others to integrate other tools into the environment if required. Indeed it was commented that "*it would be nice to have a plugin system that will enable it to be extended to other aprolog inference systems*" and to have "*the possibility to choose which solver one wants to use*".

Given that there were only a limited number of users of the other tools (at most three) it was decided not to integrate any of these tools into the environment as part of this dissertation. However, it is possible that a better response for some of these tools would have been obtained if they had also been provided as options, as respondents may have forgotten to note down that they used them. In order to clarify this, these would have to be included as part of a further questionnaire. A more detailed consideration of these tools in the future could demonstrate their advantages and justify their inclusion within the IDE. This could also have the benefit of increasing the user base of these tools.

[2] http://www.cs.utexas.edu/ tag/cmodels/
[3] http://www.mat.unical.it/ianni/wiki/dlvex
[4] http://krlab.cs.ttu.edu/ marcy/crmodels/
[5] http://assat.cs.ust.hk/
[6] http://www.cs.ttu.edu/ mellarko/aset.html

3.2.2 Target Platform

Another key requirement to consider was the platform on which the IDE would need to run. In order to develop a system that could actually be used, rather than solely for exploring features, the system would need to target the platforms used by the ASP community. This would also aid evaluations of the system, as the overhead of the subject learning to use the platform as well the IDE would be avoided, and hence eliminate this factor from comparisons with current ways of working.

The second question in the questionnaire aimed to identify the operating systems used by the community for ASP development.

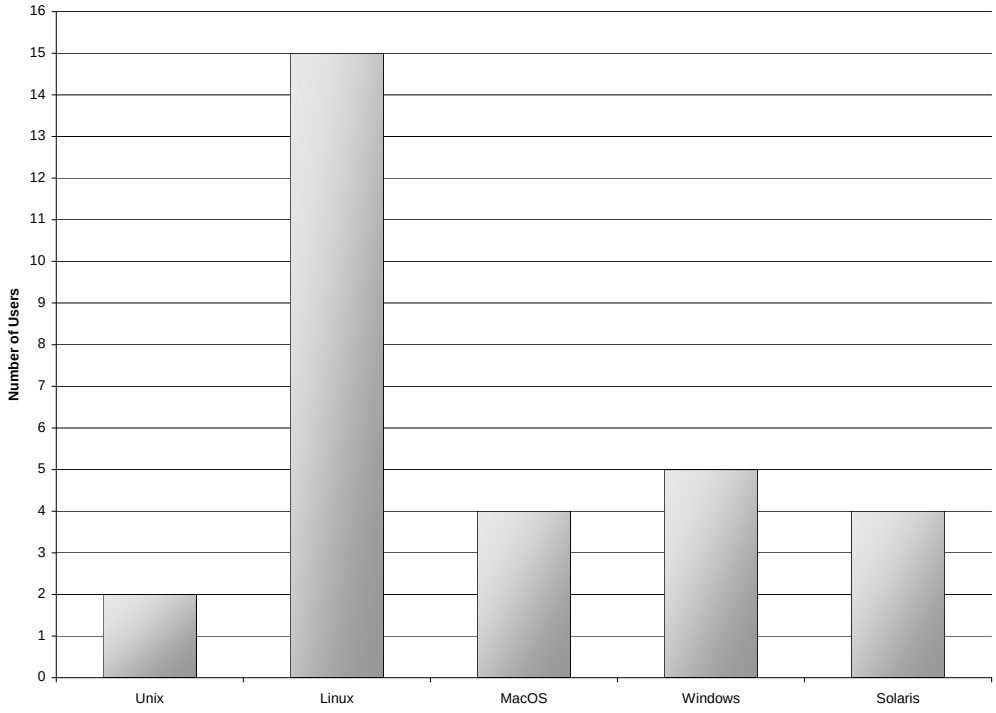


Figure 3.2: Operating Systems Used by Participants of Questionnaire

The most widely used operating system for ASP development by participants of the survey was clearly Linux (Figure 3.2). Thus this was chosen as the target platform for the system. However as other platforms were in use, and indeed some participants used these exclusively (e.g. Windows), a platform independent solution is clearly desirable.

The target platform for the IDE is also constrained by the supported platforms of any tools to be integrated. However, given that LPARSE and SMODELS are available in source form, and builds of DLV are available for Linux, Free-BSD, MacOS X and Windows this should not have a great impact on the IDE. If other tools were to be integrated that only supported specific platforms, the availability of a version for the desired platform could be arranged with the tool developer. Alternatively, a new tool could be written supporting the same functionality.

3.3 Potential Features

In order to determine some potential features for the IDE, a brainstorming session was conducted with two users of ASP tools within the Computer Science department at the University of Bath. Let us now consider why these features were judged to be of potential benefit to a user of the IDE.

Syntax highlighting could help the programmer to more easily distinguish between different elements of the program code. For example, by highlighting all keywords in a given colour it would immediately become apparent to the programmer if they attempted to use a keyword as a constant name. This error may otherwise not have been discovered until the program was run through the solver or grounder.

Providing the automatic completion of predicates (or terms) that had already been defined in the program would reduce the time taken to input the program. It would also help to reduce errors in the code caused by mistyping a name, not only by reducing the amount of typing that occurs, but equally through the lack of an expected completion indicating to the programmer that a mistake had been made.

Although the name of a predicate should be descriptive, it may not always be possible to achieve this without making the name long and cumbersome to type. Therefore it would be convenient to be able to associate a textual description with each predicate giving a more accurate definition of its meaning. However as there is currently no syntax to support this in DLV or LPARSE, this would have to be encoded within comments. If this feature was shown to be a success the inclusion of a special syntax for this could be requested from the solver developers.

Version control tools, such as the Concurrent Versions System (CVS), are often used when developing software to maintain a history of revisions of source files and facilitate several developers working together on a project. Integrating such tools into the IDE would facilitate their use within ASP development and eliminate the need to switch to an external program to interact with them.

The ability to divide a program into multiple files is important as it allows a core set of rules to be used in more than one program. For example, a set of rules encoding a problem could be defined in one file and sets of facts representing inputs to the problem in several other files. Given that input from multiple files is supported by both LPARSE and DLV, this should also be supported by an IDE.

An ASP program can be represented in terms of a dependency graph (Baral, 2003), which shows how the truth value of a predicate depends on the truth or falsity of other predicates. Providing a graph representation of programs as part of the IDE would convey this information easily, rather than having to manually extract it from the source code.

It can be difficult to find the source of errors in programs, and as discussed by Brain and De Vos (2005) this is compounded by the fact that it is difficult to determine whether an ASP program is behaving correctly. Whereas a procedural program may crash or throw an exception when an error is encountered, this is not the case for an ASP program. It would therefore be useful to include some form of debugging tools in the IDE to support this process.

When attempting to determine the reason why an answer set is not computed as expected, one aspect to consider is whether the grounding of the program is as

expected. Providing the functionality to replace a given rule by its grounding could assist the programmer in locating errors that affect the grounding process.

As discussed in section 2.4.2, constantly switching between different tools can limit the productivity of the programmer. This frequently occurs when programming, for example switching between the editor to write a program, and to the command line in order to run it. Therefore, integrating the running of the LPARSE and DLV tools and the editor into the same environment would remove this need.

Given that a complete IDE would need to support the syntax of both LPARSE and DLV, conversion between the two formats would clearly be of benefit to the users as the only alternative would be to manually rewrite the program in the other syntax.

3.3.1 Validation of Features

An important aspect of the requirements engineering process is to verify that the requirements that have been gathered for a system "*actually define the system that the customer wants*" (Sommerville, 2001). In order to understand how the features proposed at Bath would be viewed by the wider community, an informal review of the requirements was performed by presenting the list of potential features on the questionnaire. Participants were asked to rate their desire for a particular feature on a unipolar scale, with 0 being least useful and 10 being the most useful.

The results of this questionnaire have been presented as a box-and-whisker plot (Figure 3.3), in order to show the spread in the responses for each feature. The plot shows the upper and lower scores for each response (whiskers), together with the median score and interquartile range (box). Any outliers have been indicated with a cross.

From the list of features proposed on the questionnaire, it was clear that debugging tools were the most desired by the respondents, given that the majority gave this a score between 9 and 10. This would therefore be a core component of an IDE for ASP. Although there are some debugging techniques available for ASP, Brain and De Vos (2005) remark that more extensive studies into the methodology of programming in ASP are required. It was therefore not deemed viable to specifically consider debugging as part of this dissertation, although any results obtained could serve to inform the development of future debugging tools.

Another popular choice was the automatic conversion between files in the LPARSE and DLV formats. However, as the version of the IDE produced for this dissertation will only support the LPARSE language, this feature will not be implemented.

Although there was a large spread in the responses for integrating an editor with the solver, it was generally desired as most responses rated it at 5 and above. Moreover, this is an essential component of an IDE as the programmer needs to be able to edit the program and then run it through the solver. The replacement of a rule by its grounding, graph representation of programs and modularity of programs over multiple files, also appeared to be popular with the respondents as the interquartile range for each fell between a score of 5 and 8.

Furthermore, the remaining features all received a median score of at least 5 demonstrating some support for them, even if there was a wide spread in the scores that they were given. As none of the features in the list was shown to be very unpopular,

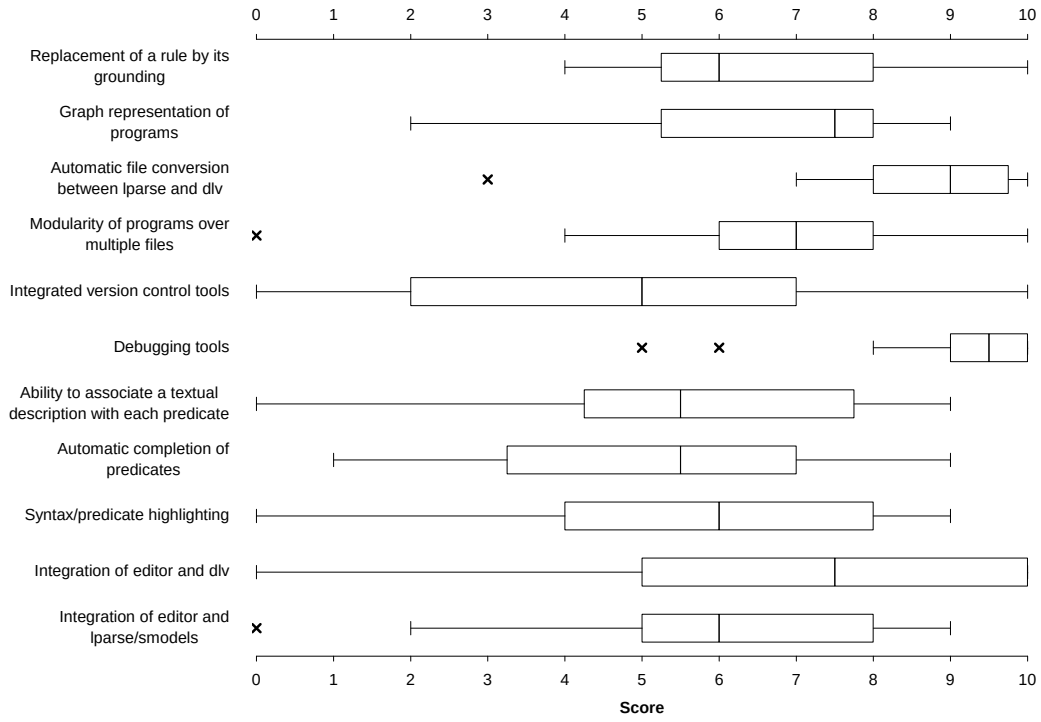


Figure 3.3: Score of Suggested IDE Features

in the way that debugging tools were shown to be very popular, it would appear that they are all judged to be of some potential by the respondents and should therefore all be explored further.

3.3.2 Suggested Features

As discussed by Preece et al. (2002), it is important to include as large a number of representatives from each stakeholder group as possible in the data gathering process, in order to avoid getting a narrow view of the requirements. Therefore in order to integrate the views of the wider community into the list of potential features, respondents to the questionnaire were asked to suggest any other features that could be included. Let us now consider these features.

One request was made to incorporate the statical analysis of program tightness into the IDE. This syntactic condition on a program is also known as positive order consistency (Babovich et al., 2000). If a program can be shown to be tight, then for that program the answer set semantics are equivalent to another semantics known as the completion semantics. In this case a satisfiability solver can be used to determine the answer sets of a program, rather than an answer set solver such as SMODELS. Including this analysis as part of the IDE could be used for indicating whether this type of solver could be used on a given program.

It was also requested to provide support for make files. Given that a program could potentially be split over several files, some of which may have already been grounded, a build script could be used to automate the process of grounding any files that had changed since last being grounded and then running the program through a solver.

The IDE would therefore need to provide support for this functionality.

In addition to this was the request to support scripts to filter the input to and output from the solvers. Providing support for scripts to perform this would permit the transformation of data from some source into a program that would be accepted by the solver, and accordingly the output from the solver to be transformed into a more useable form.

Another key feature that was suggested by one participant was automatic syntax checking. Highlighting syntax errors in the editor as they are typed, would make the error immediately evident to the programmer and prompt them to make a correction. This would eliminate the overhead of running the program through the solver before the error would be discovered, and potentially doing this multiple times to locate and correct all of the errors.

Given the range of solvers used for ASP (section 3.2.1), it was suggested that the IDE should allow the user to choose which solver they want to use when running the program. However as this dissertation is being restricted to supporting the LPARSE and SMODELS tools, this feature will not be considered. Related to this was the ability to provide benchmarks for the different solvers in the system, such as the time taken to run the solver. This feature would allow the user to compare different solvers and potentially choose the one most suited to their specific program.

Other features that were suggested could be used to support the debugging of ASP programs. In their query based debugging approach, Brain and De Vos (2005) identify that both the presence of unexpected atoms in an answer set, or lack of expected atoms are indicators of an error in the program. In order to identify the source of the error, the programmer needs to understand why a given atom is either present or excluded from an answer set. Both the ability to provide a "*tracing facility for computing models bottom up*", and the ability to display the rules used to generate an answer set, would help the programmer to understand why a particular atom is included in an answer set and thus support this process.

The value of generating the dependency graph for a program has already been considered, however this was reiterated with requests to display the components of these graphs (such as the atoms) and the dependencies between them.

3.4 Conclusion

This initial requirements gathering phase has helped to identify some of the non-functional requirements of the system, such as the platform on which it must operate and the tools that it must support. It has also produced a list of potential features that could be included in the system, and should therefore be explored in more detail. However, given the time constraints of the dissertation it was not intended for all of these features to be explored. The requirements from this stage of the elicitation process that are considered in this dissertation are summarised below:

- Support for LPARSE and SMODELS tools
- Multi-platform support
- Syntax highlighting
- Automatic syntax checking

- Integrated version control tools
- Multiple file support
- Display of program dependency graph
- Integration of editor and LPARSE
- Integrated build script support

Chapter 4

Eclipse

After gathering the initial requirements of the system, it was chosen to develop the IDE as a plug-in for the Eclipse platform. This chapter outlines the rationale for choosing this approach and discusses how the Eclipse plug-in architecture influenced the overall system design.

4.1 Choice of Development Approach

4.1.1 Plug-in Architecture

It was demonstrated in the requirements analysis that the system should provide an extension mechanism to allow other solvers and tools to be integrated into the system in the future (section 3.2.1). As discussed in section 2.4.3, Eclipse is designed as a set of plug-ins and provides the PDE to facilitate the development of new plug-ins. A system developed as an Eclipse plug-in could therefore be extended by other ASP users to support their own specific tools. Using the plug-in mechanism already provided by Eclipse also alleviates the need to design and implement this functionality, which would have been necessary had the system been developed from scratch.

Moreover, the dissertation is following an exploratory development process, which according to Sommerville (2001) can lead to systems becoming poorly structured. By developing the system as an Eclipse plug-in, the design of the system would have to fit with this existing architecture. Thus this approach would impose some structure on the system, constraining any maintenance issues arising from the rapid development to the plug-in itself. Whereas had the plug-in been developed from scratch, these could have affected the entire system.

4.1.2 Development Cost

One of the key reasons for choosing to develop the IDE as an Eclipse plug-in was that the platform already provided a lot of basic functionality.

Amongst the functionality already present in Eclipse is a text editor, which is clearly a fundamental tool for writing ASP programs. The editor already implements the

loading and saving of files, together with common functionality such as undo/redo, copy/cut/paste, find/replace, line numbering and printing. In addition to this, the Eclipse API provides classes to support the implementation of syntax highlighting, annotations, autocompletion and custom actions. This would facilitate the development of ASP-specific editor features and reduce the cost of their development.

Furthermore, Eclipse incorporates a launching mechanism that allows external tools to be launched from within the IDE, together with a console for any textual input or output. This would allow LPARSE, SMODELS and DLV to be launched from within Eclipse as well as any other external ASP tools. The platform also includes two of the features identified in the requirements analysis: a CVS client is integrated into the environment as well as support for Ant build scripts.

Other functionality provided by the platform includes resource management tools to group files into projects, browse the files in the workspace, open external files, maintain a local file history, file comparison and the ability to search across multiple files. Should any other 'standard' functionality also be required by the system, there is also a very extensive base of third party plug-ins to draw upon. Indeed at the time of writing, the Eclipse Plug-in Central[1] website listed 506 plug-ins, and the EclipsePlugins[2] website 1177.

Given the functionality that is already present in Eclipse, it would be possible to use this framework to write AnsProlog* programs and run them through a solver without needing to write any additional code. This approach is adopted in the development of the proof of concept system (section 5.1). However in order to fully integrate the tools into the environment, some development would still be required, although this would be significantly less than the cost of developing the entire system from scratch. As discussed in section 2.4.3, adopting this approach would allow more time to be devoted to developing ASP specific tools and features rather than re-implementing standard functionality.

4.1.3 Multi-platform Support

The requirements analysis showed Linux to be the key operating system that needed to be supported by the IDE. However, it was also desired that the system be platform independent in order to accommodate users of other operating systems.

The majority of the Eclipse code is written in pure Java. Therefore this part of the platform does not depend directly on the underlying operating system, rather on the availability of a Java 2 platform for that operating system (Eclipse, 2005b). Given that plug-in code is also written in Java, the ASP plug-in would only be dependent on the Java 2 platform, providing that no platform specific code were included. This would therefore help to reduce the cost of developing and maintaining the system as it would not be necessary to maintain different versions of the code for each operating system. Nevertheless, it would still be important to test the system on the various platforms to ensure that no dependencies on a specific platform would be introduced.

However, the platform does still have some platform specific components, such as

[1] http://www.eclipseplugincentral.com
[2] http://eclipse-plugins.2y.net

the Standard Widget Toolkit (SWT) UI library and Eclipse executable. A source distribution of the Eclipse platform is available to allow it to be ported to more platforms (Eclipse, 2005c), although it is noted that this may require writing patches for the launcher and SWT libraries. Despite this potential drawback, releases of the Eclipse platform are available for "*Windows, Linux, Solaris, HP, Mac OSX, and others.*" (Clayberg and Rubel, 2004) and can therefore be used on the platforms identified in the requirements analysis.

4.1.4 Other Extendable Systems

The previous sections have outlined how the plug-in architecture and multiplatform support provided by Eclipse, together with their associated reduction in development cost, make this platform suitable for the development of the IDE. Let us now consider some other systems that meet these requirements and discuss why Eclipse was chosen in preference to them.

In section 2.4.3 the existence of an Emacs mode for SMODELS was identified, and it was discussed that a potential development approach for the IDE would be to extend this. Indeed Emacs also provides many of the benefits that we have just considered for Eclipse: an extension-based architecture, support for multiple platforms and some existing common functionality. As Emacs is based on the Lisp programming language (Glickstein, 1997), experience of development using this language would have been important to facilitate the rapid development of the system. As the author has more experience in development using Java, the language used by Eclipse, it was felt that this approach would allow a greater number of features to be explored within the time constraints of the dissertation.

Another similar system to Eclipse that was considered to be used as a basis for this system was the NetBeans[3] platform. Like Eclipse, this platform is also well known as a Java IDE (Geer, 2005), and has a similar plug-in architecture to Eclipse using a mixture of Java and XML and providing its own tools to support the development of plug-ins (NetBeans, n.d.). However, one area in which this would have an advantage over Eclipse for implementing the ASP IDE is in the area of portability. The entire IDE is written in pure Java and is therefore only dependent on the availability of the Java platform (Vaughan-Nichols, 2003), whereas as we have just considered, Eclipse has some direct dependencies on the underlying platform including the SWT library. A plug-in written for this system could therefore be used on a larger number of platforms than one written for Eclipse. However, Vaughan-Nichols also notes that these platform dependent UI elements have been observed to give Eclipse a better performance than NetBeans, and given that the tools integrated into the IDE would also have direct platform dependencies this was not considered to be a sufficient reason for choosing this approach.

Moreover as we have considered in section 2.4.3, Eclipse seems to be a popular choice of IDE at present with a "*growing buzz*" surrounding it (Wolfe, 2003). Indeed, in comparison to the hundreds of plug-ins listed for Eclipse, the NetBeans plug-in catalogue[4] only listed 47 at the time of writing. Support to this view is also given in a recent poll by LinuxQuestions.org (2006), in which Eclipse was voted IDE of

[3] http://www.netbeans.org/

[4] http://www.netbeans.org/catalogue/all-stable.html

the year 2005. It received 32% of the votes, whereas Emacs and NetBeans received 13% and 6% of the votes respectively. Given the current popularity of this platform, especially with users of the primary target operating system, it was felt that adopting this approach would safeguard the longevity of the IDE.

4.1.5 Acceptance by the ASP Community

Another important factor to consider when choosing the development approach was whether a product based on the Eclipse platform would meet the needs of the ASP community. That is to say, validate that the proposed solution meets their needs, and if not, identify further requirements of the system that would guide the choice of a more appropriate development approach. Indeed if this validation were not performed, then the resulting system would potentially not be used in practice and thus prevent the system from being used as a mechanism for identifying ASP tool requirements.

Let us now consider how the reaction of the community to this approach was determined, through a questionnaire and demonstration of a prototype system.

Questionnaire

Given that the development of the IDE as a plug-in for Eclipse had been identified as a potential development approach at the beginning of the project, part of the questionnaire sent out to the ASP community (Appendix B) was dedicated to assessing whether this would be an acceptable approach. The questions aimed to determine how many of the participants were aware of the platform, identify any potential problems with using this tool and suggest any alternative approaches.

Only 5 participants responded that they were familiar with the Eclipse platform (7 were not familiar and 5 did not respond). However, the responses indicate that this question may have been misinterpreted, as one participant who indicated that they were not familiar with the platform commented that they had used it "*a long time ago*". The wording '*Are you aware of the Eclipse development platform?*' may have been more appropriate.

However, out of those that indicated that they were familiar with the platform there were no objections to its use as a foundation for the IDE as long as the resulting system "*works well and is simple to use*". Indeed one participant commented that it would be interesting to use this approach "*since it is a widely known framework and a lot of plug-ins already exist that may be exploited*".

Only one respondent suggested Emacs as an alternative approach, but conceded that "*Eclipse seems to be the way of doing things these days*". The only concern raised was whether this would necessitate the use of non-free tools, which is not the case.

Although few respondents were familiar with the system, the lack of objections to this approach are encouraging given that the question aimed to draw out any potential problems. However, a greater number of results would clearly be required in order to draw any firm conclusions from this.

Demonstration of Prototype

Following the initial requirements analysis, a basic prototype system was developed to help draw out further requirements from users of ASP within the department. Furthermore, the demonstration of this prototype would also help to identify any concerns with this approach. Although the reaction to the prototype was positive, the demonstration of the system to a larger group of users would have been more likely to draw out potential issues. However this was not possible within the constraints of the project.

The development of the prototype system, and the requirements obtained from its demonstration, are discussed in more detail in the next chapter.

From the validation that could be performed within the constraints of the dissertation writing a plug-in for Eclipse does appear to be a suitable development approach for the IDE. Moreover, as this approach was accepted by the users at the university to which the prototype was demonstrated, this would still permit the system to be used to explore the requirements of the IDE with these users, even if a different implementation technique would be required to develop a final system that would be acceptable to the wider community.

4.2 Architecture

Choosing to develop the IDE as a plug-in for Eclipse, constrains the overall design of the system to fit with the Eclipse plug-in architecture. In the latest versions of Eclipse (3.x) this "*is based on technology by the OSGi Alliance*"[5] (Clayberg and Rubel, 2004).

An Eclipse plug-in consists of a Java Archive (JAR) File containing the compiled plug-in code, any resources and two configuration files (Eclipse, 2005a). The manifest.mf file contains information regarding the name and version of the plug-in as well as its dependencies on other plug-ins. The plugin.xml file defines the *extensions* that the plug-in makes to other plug-ins, as well as any *extension points* which other plug-ins can extend. As part of its code, each plug-in must also provide a class that extends the `Plugin` class, in order to provide access to resources, preferences and other information (Clayberg and Rubel, 2004).

Plug-ins are installed in the *plugins* subdirectory of the Eclipse installation. When the Eclipse platform starts, it reads the manifest files of all the available plug-ins and stores this information in an internal model known as the plug-in registry (Eclipse, 2005a). At this point the plug-in code itself is not loaded, this only occurs when the plug-in is actually needed. This reduces the memory footprint of the platform at start-up, given that some plug-ins may not be used during a given session.

Rather than supply the IDE as a single plug-in, its functionality has been divided into several plug-ins: a main plug-in for common ASP functionality, a dependency graph plug-in and separate plug-ins for the LPARSE and SMODELS tools. The IDE was separated in this way to allow other developers to build upon the IDE, using only the features they need. For example, a plug-in for a solver other than SMODELS

[5] http://www.osgi.org/

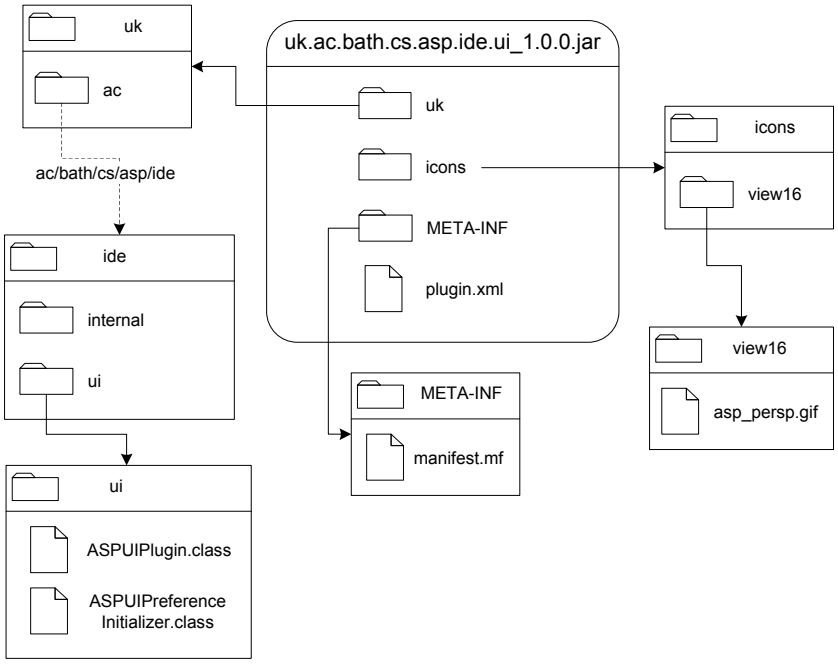


Figure 4.1: Eclipse Plug-in Structure

could use the functionality of the LPARSE plug-in, without requiring the SMODELS functionality to be installed.

Commercial plug-ins are also often further subdivided into Core and UI plug-ins, to separate plug-ins that can operate in a headless environment from those that depend on a user interface (Clayberg and Rubel, 2004). This separation would enable another plug-in developer to utilise some of the functionality of the system (e.g. launching LPARSE, parsing source files) without having to include the UI components. Thus each of the plug-ins described above, have also been divided into core and UI components.

The functionality included in each of the plugins are as follows:

- uk.ac.bath.cs.asp.ide - Logging
- uk.ac.bath.cs.asp.ide.ui - ASP Perspective, Syntax Coloring Preferences
- uk.ac.bath.cs.asp.ide.dependencygraphs - Dependency Graph Model, Dependency Graph UI
- uk.ac.bath.cs.asp.ide.lparse - LPARSE Preferences, LPARSE Parser and Document Model, LPARSE Launcher
- uk.ac.bath.cs.asp.ide.lparse.ui - LPARSE Editor, LPARSE Preference Page, LPARSE Launch Configuration UI
- uk.ac.bath.cs.asp.ide.smodels - SMODELS Preferences, SMODELS Launcher
- uk.ac.bath.cs.asp.ide.smodels.ui - SMODELS Preference Page, SMODELS Launch Configuration UI

In order to manage groups of plug-ins that can function as a single unit, Eclipse also

includes a feature framework (Clayberg and Rubel, 2004). This facilitates branding a feature, providing licensing information, packaging and deploying the feature via an update site and managing the versions of the constituent plug-ins. The feature manifest, feature.xml, defines the plug-ins that make up the feature and contains the licensing information. This is installed to the *features* subdirectory of Eclipse. Any branding files are included in the root of the main plug-in for the feature (uk.ac.bath.cs.asp.ide). The individual plug-ins of the IDE have therefore been grouped together as the AnsProlog* Programming Environment (APE) feature.

4.2.1 Extension Points and Extensions

In order that other plug-ins may extend the functionality provided by a plug-in, in a loosely coupled way, that plug-in can define an extension point (Clayberg and Rubel, 2004). The plug-in defines a minimal set of classes and interfaces to be used with the extension, together with a schema defining how the extension point should be used. An example of this is the `org.eclipse.ui.views` extension point, which allows new views to be added to the workbench.

Although there are no extension points defined in this dissertation, they should be used in the final system to provide a structured mechanism for other developers to integrate further ASP tools into the IDE, including additional solvers as specified in the requirements. These were not considered as the exploration of the requirements of the IDE through system usage was constrained to users within the department. Therefore this feature would be unlikely to be used until the system achieves a wider user base.

The system does make use of extension points, however, to extend the existing functionality in Eclipse. The design of the system is therefore constrained by the API defined by the plug-in being extended. For example, in order to contribute the dependency graph view to the plug-in (section 6.3.3), an extension to `org.eclipse.ui.views` has to be defined in plugin.xml together with a class implementing the `IViewPart` interface.

```
<plugin>
   <extension
         point="org.eclipse.ui.views">
      <view
            category="uk.ac.bath.cs.asp.ide.views"
            class="uk.ac.bath.cs.asp.ide.dependencygraphs.ui.views.
                   DependencyGraphView"
            icon="icons/eview16/depgraph_view.gif"
            id="uk.ac.bath.cs.asp.ide.dependencygraphs.ui.views.
                DependencyGraphView"
            name="Dependency Graph"/>
   </extension>
</plugin>
```

4.2.2 Internal Code

Any classes written for a plug-in are separated into classes that are to be used internally by the plug-in, or are made public API in order that they may be used by other plug-ins (Clayberg and Rubel, 2004). The list of packages that are 'exported', in other words made public API, are defined in the manifest.mf. The convention of including the term 'internal' in the name of any packages that contain internal code is followed in this dissertation. Although no extension points have been defined in the system implemented, any code that has been designed to be reused by other plug-ins is made available as public API.

4.2.3 User Interface

Another area in which Eclipse constrains the design of the system is in the user interface. Eclipse uses the SWT library which provides "*a thin compatibility layer on top of the platform's native controls*", rather than Sun's Abstract Windowing Toolkit (AWT) and Swing libraries that come with Java (Clayberg and Rubel, 2004). In addition to this, Eclipse provides the JFace library which wraps the basic widget classes to facilitate the transformation of data between its object oriented representation and the simple data types provided by the native widgets. Therefore in order to remain consistent with the rest of the IDE, the plug-in had to use these libraries for the user interface.

As well as these libraries, Eclipse defines a set of user interface guidelines that plug-ins should adhere to (Edgar et al., 2004). Although the guidelines have been attempted to be followed for the IDE, there was insufficient time to perform a rigorous verification that the system conformed to all of these points. However it is important that a full system should be checked for conformity to the guidelines, as otherwise it may not give the user the impression of being integrated into Eclipse. An example that has been followed in this dissertation is the palette, size, placement and naming conventions used for any icons (guidelines 2.1 - 2.21).

4.3 Conclusion

Given the requirements gathered in the previous section, it was chosen to develop the IDE as a plug-in for Eclipse, as this met the requirements for the system to run on multiple platforms and be extendable. The provision of existing functionality makes it suitable for rapidly developing a prototype system, and the plug-in architecture enforces an underlying structure on the evolutionary system being produced in order to aid future maintenance. It was also chosen over similar systems such as Emacs, to eliminate the overhead of learning a new language when attempting to develop the system rapidly. Given the current popularity of the platform, it was deemed better to support this technology than the less favoured options in an attempt to avoid extending a tool that could become obsolete.

This development approach was validated with both the wider community through a questionnaire, and users within the department with a demonstration of a prototype. Although the development approach was not rejected, it was felt that a demonstration of the system to the wider community would have given a better

indication as to whether this was indeed suitable. However as this was not possible within the constraints of the project, it was decided to use this approach.

The Eclipse architecture imposed a number of constraints on the design of the system with respect to the plug-in structure, naming conventions and the user interface libraries and design. Although full conformance to the user interface guidelines could not be comprehensively checked within the timescale of the dissertation, this would need to be performed on any final system delivered to the users. Similarly extension points would need to be defined in the final version, in order to provide a more structured extension mechanism for the features implemented than just the reuse of code declared as public API.

Chapter 5

Prototype System

Prior to designing and implementing the actual IDE, a prototype system was developed. Sommerville (2001) observes that prototypes can be used to elicit further requirements from the users, as well as validate that the existing requirements are correct. Therefore this technique was chosen to improve the initial requirements specification obtained from the brainstorming session and questionnaire, and validate that an Eclipse based system would meet the users' requirements as described in section 4.1.5.

Preece et al. (2002) categorise prototype systems into two classes. The first, low-fidelity prototypes, look little like the final system and are often made of different materials, for example a paper based sketch. Because these systems are quick and inexpensive to develop and modify, they are useful for exploring different design approaches. High-fidelity prototypes look more like the final system, but are therefore often costly and time consuming to develop which can make them ineffective for exploring requirements.

Let us consider how these approaches were used to first implement a proof of concept system featuring basic syntax highlighting and launching of LPARSE and SMODELS, and then develop this into a higher-fidelity prototype that would be demonstrated to the users.

5.1 Proof of Concept System

The initial proof of concept system developed was situated between the high and low fidelity approaches, and combined their benefits. As it was proposed to develop the system as an Eclipse plug-in, there were many existing components that could be reused to rapidly develop the system as described by Sommerville (2001). The system produced was therefore cheap to implement and so little development effort would have been wasted if the Eclipse based system was rejected. Furthermore, as the system was a set of Eclipse plug-ins it looked very similar to the actual system, and could therefore be used to validate this approach. Let us now examine how the system was implemented.

5.1.1 Perspective

In Eclipse, perspectives are used to define the layout of views and actions that are available for performing a given task (Clayberg and Rubel, 2004). A new perspective was therefore defined for the task of ASP development. The initial version was defined with the Navigator view on the left for browsing files in the workspace, the Editor in the centre, and the Console view below for displaying solver output (Figure 5.1). This was implemented by defining a new extension to the `org.eclipse.ui.perspectives` extension point and creating a class implementing the `IPerspectiveFactory` interface.

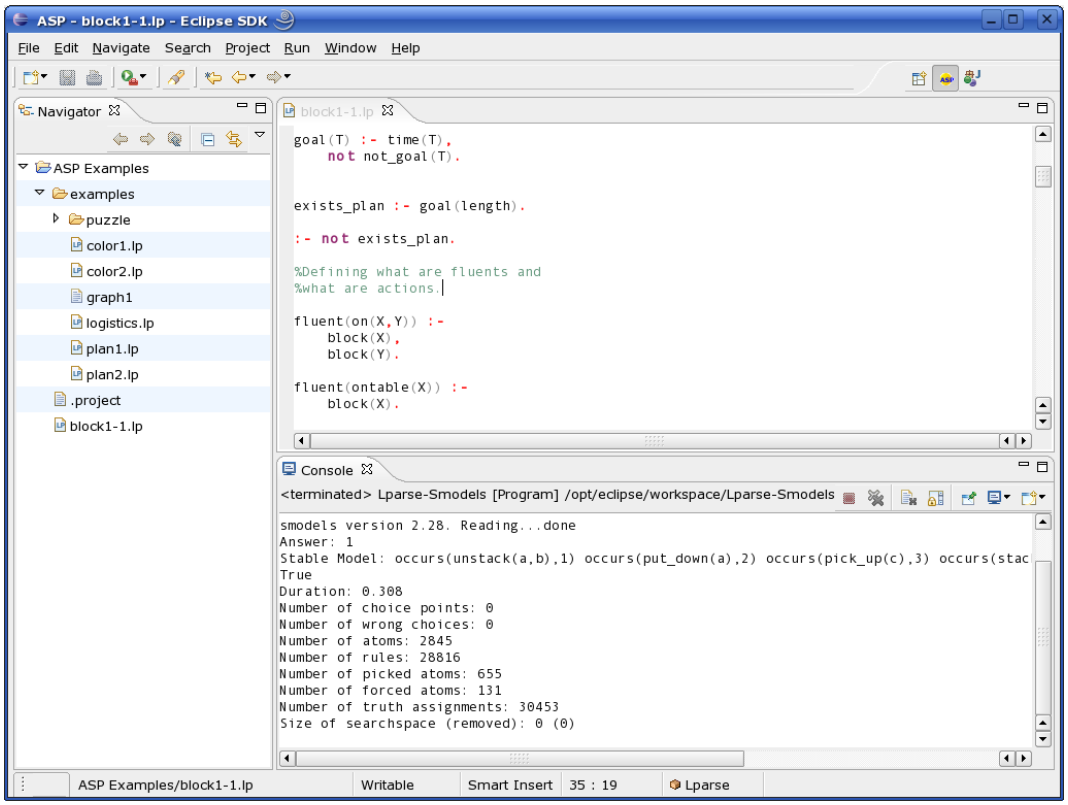


Figure 5.1: Proof of Concept System

5.1.2 Syntax Highlighting

In order to rapidly develop the syntax highlighting for the LPARSE editor, the initial system utilised the Gstaff ColorEditor plug-in[1]. This plug-in configures the syntax highlighting of the Eclipse text editor using JEdit's syntax highlighting definitions. The plug-in also provides a preference page for setting the colour of each element and whether the font is emboldened.

In order to support syntax highlighting for LPARSE source files, a new syntax highlighting mode was written based on existing modes for other languages. This provided syntax highlighting for LPARSE keywords, comments, built-in functions and operators. The rules to highlight variables worked successfully under JEdit, but not

[1] http://www.gstaff.org/colorEditor/

under the syntax coloring editor plug-in. However this was not deemed important for the proof of concept system. Highlighting other names such as predicates and constants was also not possible as this would have required parsing the source file to determine the type of each identifier in the file. Syntax highlighting is discussed in more detail in section 5.2.1.

In addition to creating the new mode, the existing plugin.xml file had to be modified to register the LPARSE file extension (*.lp) with the ColoringEditor. The ColoringEditor's file icon was also replaced with an icon for LPARSE files.

5.1.3 Launching

The ability to launch LPARSE and SMODELS from within Eclipse was implemented with a batch file on Windows and shell script on Linux. This batch file was created in a separate project in the workspace, and the navigator view was filtered to make this transparent to the users.

An external tools launch configuration was created within Eclipse to launch this script and display the output in the console view. The file to run was supplied as an argument to the batch file, namely the currently selected file in the resource navigator. The locations of the LPARSE and SMODELS executables and any additional arguments were passed as environment variables.

5.2 Development of Prototype

In order to evaluate the proof of concept system, a demonstration of the system to the users was organised. However as the development of the proof of concept system was completed a week before the session, it was decided to enhance it by implementing the plug-in code for the syntax highlighting and launching functionality. Although this extra development effort could have been wasted if the Eclipse based approach had been rejected by the users, it was considered to be good use of the time available before the demonstration session.

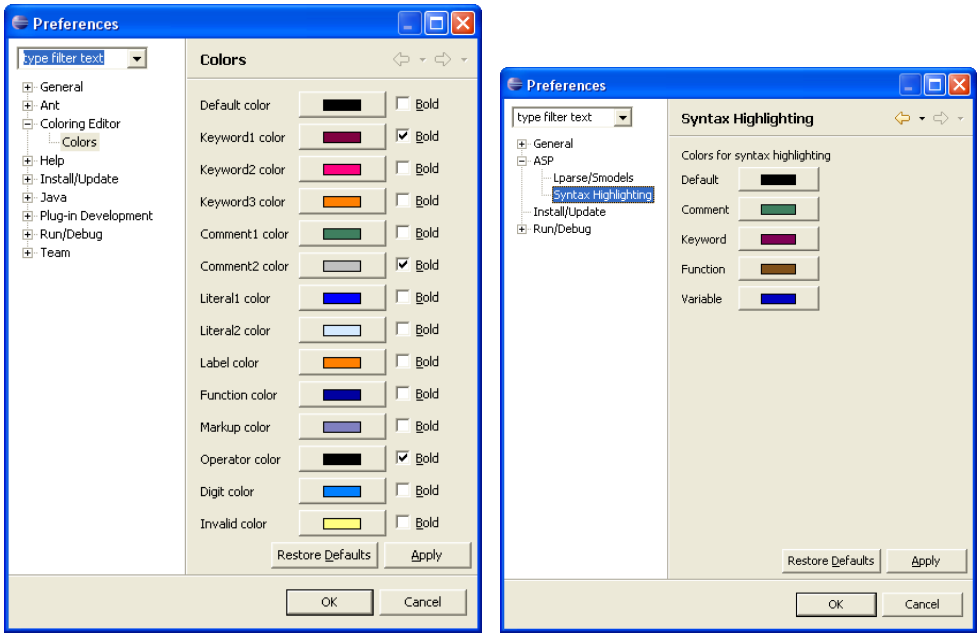
The aim of this extra development was to produce a user interface that would be closer to the final system, before the demonstration session. This would allow users to provide useful feedback on the proposed user interface, rather than the crude implementation in the proof of concept system. Moreover the development of this higher fidelity system also served to gain a familiarity with the Eclipse API relevant to this functionality. If the approach was accepted, this would enable these aspects to be implemented more quickly in the final system.

5.2.1 Syntax Highlighting

The improvements made to the syntax highlighting element of the proof of concept system involved replacing the ColoringEditor plug-in with dedicated plug-in classes for LPARSE. In the proof of concept system, any files that had an associated JEdit mode could be opened with the editor. As the icon for the editor had been replaced with an LPARSE file icon, all files that could be opened with the editor would be

assigned the same icon. This could have potentially confused the user during the demonstration.

In addition, the preference page provided by the editor (Figure 5.2a) lists all of the syntax categories provided by JEdit. Given that not all of these categories are required for ASP, and that some of these categories (e.g. Literal) have a specific meaning in ASP, this could have confused the user. Therefore, a new dialog was created incorporating only the elements that would be highlighted (Figure 5.2b), and this was placed under an ASP category for ease of location.



(a) Proof of Concept System (b) Prototype System

Figure 5.2: Syntax Highlighting Preference Dialogs

Let us now consider which elements of the LPARSE syntax were chosen for highlighting, and why these were chosen.

The syntax of many programming languages contain keywords that can only be used in specific contexts, for example not as a variable name, and LPARSE is no exception (Syrjänen, n.d.a). A common feature of syntax highlighting, is to highlight these keywords in a different colour or style to the rest of the code, and indeed this is the case in the JDT. This helps the programmer to identify that they have entered a keyword, aiding the detection of errors caused by keywords used in an incorrect location (text is unexpectedly highlighted), or the misspelling of keywords (text is not highlighted as expected). It was therefore decided to use this concept in the IDE's highlighting scheme, and use the same highlighting style as used in the JDT to maintain consistency on the platform.

Comments are also commonly highlighted in order to visually separate them from the functional contents of the source file. This was therefore also included in the highlighting scheme for the IDE, and again maintained the same style as the JDT.

Another concept that is used by the JDT's syntax highlighting scheme is to highlight certain types such as strings, and properties of certain members such as their scope and whether they are constant. However the concept of differing scopes is not

applicable to ASP, and thus is not used in the highlighting scheme. What is useful to consider, however, is that AnsProlog* programs contain only a limited number of types (variables, constants, functions, atoms) as described in section 2.2. It was felt that it may be useful to highlight these elements in order that they could be quickly distinguished from one another, for example whether a symbol is a function or a constant. We now consider the implementation of the syntax highlighting and why this was restricted to variables and built-in functions for the prototype.

Implementation

In order to develop the syntax highlighting and allow the future development of other LPARSE specific editor features, a custom editor for Eclipse was developed following the tutorial by Ho (2003). This was implemented as a subclass of the `TextEditor` class in order to inherit all of the common text editor functionality provided by Eclipse. In order for the new editor to be recognised by the platform, an associated extension entry was also created in plugin.xml.

The editor provides a `SourceViewer` which is responsible for managing the pluggable behaviour of the editor (Eclipse, 2005d). However, this does not by default provide support for syntax highlighting, as it is not aware of the structure of the document that it is displaying. In order to configure this behaviour for LPARSE, a subclass of the `SourceViewerConfiguration` class was implemented, and the `getPresentationReconciler` method overridden to provide a custom `PresentationReconciler` for the `SourceViewer`.

Eclipse uses the damage, repair and reconcile model to handle the updating of syntax highlighting. In this approach the text that needs to be redisplayed is computed and marked as damaged, and the appropriate repairs are applied to the text. This process is called reconciling. The `PresentationReconciler` created for the editor was configured with a class to handle both the damaging and repairing of the text - a custom subclass of the `RuleBasedScanner` class.

Wilhelm and Maurer (1995) describe a scanner as the module responsible for the "*lexical analysis of a source program*". This transforms the input program into a sequence of units called tokens, with tokens classified into a "*finite set of token types*"(Appel and Ginsburg, 2004). A `RuleBasedScanner` object is given a set of rules which it uses to generate the tokens for the source file (Eclipse, 2005d). It is also configured with a set of `TextAttributes` defining the colour and style to apply to the text corresponding to the token.

The scanner specification for the LPARSE tool, available in the source distribution, defines the different token types that are recognised by LPARSE. The rules for the custom `RuleBasedScanner` were therefore based upon the definitions used in this file, in order for the editor to correctly recognise the various tokens. However, given that the identifier token is used for the names of constants, functions and atoms, it was not possible to ascertain which type a given identifier represented using simply a scanner. This would require performing additional syntactic and semantic analysis of the source file in order to determine its meaning (Appel and Ginsburg, 2004).

Given the additional time required to design and implement this, it was not considered for the prototype system. Thus the elements highlighted by the system were limited to keywords, comments, built in functions (as if an identifier matched one

of these names it was known to be a function) and variables.

5.2.2 Launching

Although the mechanism of launching SMODELS in the proof of concept system would have been transparent to a user attempting to perform a basic launch of the tools, this would not have been the case had they needed to configure the command line. In order to do this, they would have needed to edit the environment variables in the external tools configuration. Therefore, to facilitate this process a dedicated user interface was developed.

The locations of the executable files were moved to a preference page under the ASP category, as making this a global setting would allow the tools to be accessed from other parts of the system if required. The location fields used the `FileFieldEditor` component, which automatically handles the loading and saving of the preference values, provides an interface to allow the path to be entered manually in a text field or through a file selection dialog, and validates that the specified file does exist. This made this aspect very quick to develop.

The launching framework in Eclipse defines the concept of launch configuration types and launch configurations (Szurszewski, 2003). A launch configuration type defines how to launch a configuration and specifies the parameters required to perform the launch, whereas the launch configuration defines the values of the parameters. For example, the launch configuration written for this prototype specifies how to run LPARSE and SMODELS given a set of parameters. When the user wishes to launch these tools, they create a new configuration specifying the desired parameters.

Launch configuration types need to be declared with an extension entry in plugin.xml, and implemented with a class that implements the `ILaunchConfigurationDelegate` interface. This only requires one method to be implemented, the `launch` method that performs the launch given the supplied configuration. In this method, a standard Java `Process` is created to run the desired tool and this is wrapped in an Eclipse `RuntimeProcess` to allow this to be use by the launching framework.

However, the launch of SMODELS required the output of LPARSE to be piped to the input of SMODELS. Therefore separate processes were created for both tools, with a loop to read the contents of the LPARSE output stream and write this onto the SMODELS input stream. As the output of SMODELS needed to be displayed in the console view, this was wrapped in the `RuntimeProcess` object.

The launching framework provides a user interface for managing launch configurations - the launch configuration dialog (Figure 5.3). Each launch configuration type can be assigned its own tab group comprised of a set of tabs with controls for editing the launch configuration. The SMODELS tab group was defined as an extension in plugin.xml, and implemented as a subclass of `AbstractLaunchConfigurationTabGroup`. This included a tab for editing the LPARSE and SMODELS arguments and the `CommonTab`, which should be included in all tab groups and by convention is the last tab in the group. The SMODELS tab provided text fields for listing the options, rather than having separate controls for each option, in order to develop the prototype system quickly.

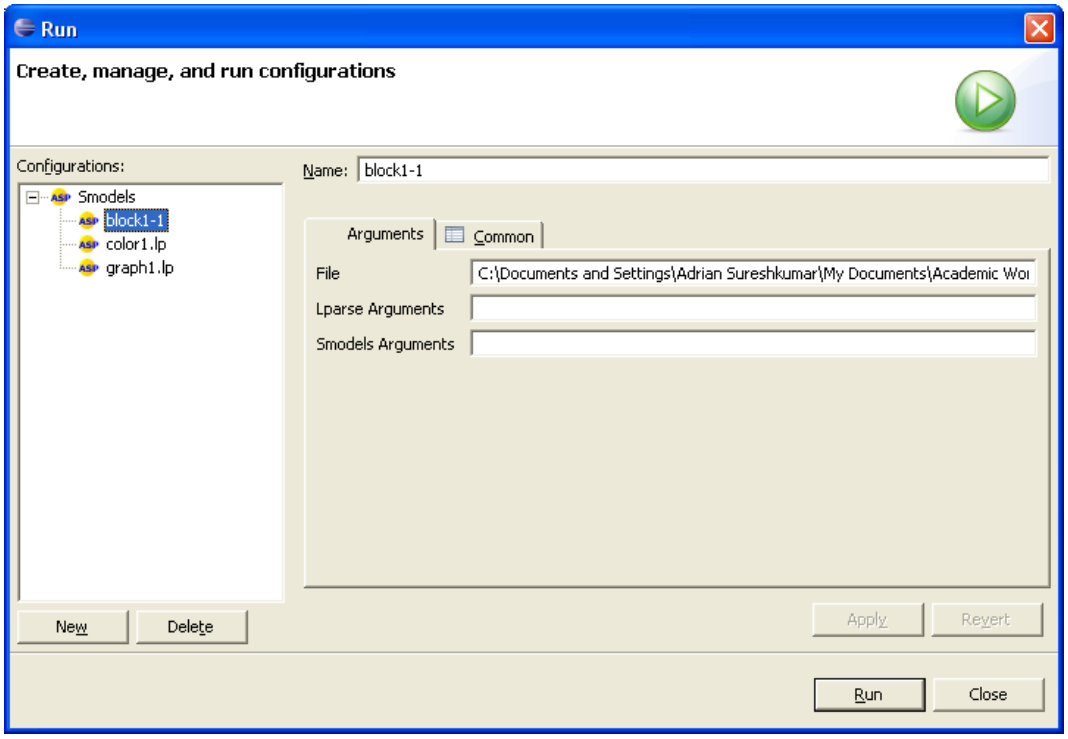


Figure 5.3: Prototype Launch Configuration Dialog

The framework also allows the creation of launch shortcuts, which are used to create a default launch configuration from the current selection or current editor, and perform the launch. These can be activated by the user from the Run menu. For the prototype, a shortcut was implemented to create a new launch configuration from the first file in the current selection in the resource navigator, or use the existing one if it existed. Although the full system should support multiple files, this was deemed to be sufficient for demonstration purposes.

5.3 Demonstration of Prototype

As previously mentioned, a demonstration of the completed prototype was organised with the ASP users within the department in order to evaluate the system. Although this was only attended by three of the five invitees, it still provided useful feedback on the prototype system and other potential features to be included in the IDE.

The session consisted of an overview of the Eclipse platform before the demonstration of the ASP plug-in. This aimed to provide the users that were unfamiliar with Eclipse, with a basic understanding of the platform in order that they could relate the features of the plug-in to what was already provided by the platform. During the demonstration of the plug-in itself, participants were asked for their views on each of the features in order to encourage feedback on the system. Following this an overview of the features of the Java Development Tooling (JDT) plugin was given, in order to determine whether any features that had been applied to this plug-in could be applied to ASP.

Given the small size of the group, the demonstration was presented on a laptop with

the group seated around a desk, rather than projected onto a screen in a lecture theatre. This informal setting facilitated the interaction between the presenter and the users, from which the feedback was gathered. In retrospect, had more users been available then it may have been appropriate to separate them into small groups and use the same approach.

Ideally the prototype system would have been distributed to the wider ASP community in order to gain feedback from a more diverse user group. However this was not achieved, given that the cost of developing user documentation to support the evaluators effectively and of developing a suitable licensing agreement was not deemed acceptable for an initial prototype system. Nevertheless, it is intended that the final system will be released to the community following the submission of this dissertation in order to guide any further work.

An alternative approach would have been to perform the demonstration to these users, as this would alleviate the cost of developing the documentation and license for the system. However, this was again not viable due to the cost of travelling to the relevant institutions to perform the demonstration.

5.3.1 Editor Feedback

As we have just considered, the syntax highlighting in the prototype only highlighted keywords, comments and variables. The demonstration indicated that although variables must begin with an upper case letter and constants with a lower case, it would be useful to highlight constants in a different colour to facilitate this distinction further. It was also requested to highlight the `:-` operator, as well as the `.` character which is often missed at the end of a line. Another suggestion was to highlight a negated atom in a different style, to emphasis the context in which it is used. This should be similar for classical negation and negation as failure.

It was suggested that the editor should be able to provide automatic indentation of LPARSE source files, in order to reduce the effort required by the developer to maintain their code in an easy to read format. However in order to determine the layout scheme to which the indentation should adhere, an investigation into coding standards for ASP would need to be performed. This would be beyond the scope of this dissertation.

Given that LPARSE programs contain predicates, which may in turn contain nested functions, a rule may contain several layers of brackets. Providing bracket matching functionality would help to indicate to the programmer which bracket was closed, and thus prevent errors caused by closing brackets in the wrong place, for example:

```
happy(J) :- eats(J, bananas, plus(times(8, plus(7,3))), minus(7,5)).
happy(J) :- eats(J, bananas, plus(times(8, plus(7,3)), minus(7,5))).
```

In addition to automatically checking the syntax of the source file and highlighting these errors in the editor, as discussed in the requirements (section 3.3.2), it was suggested to highlight any warnings that can be output by LPARSE. This would enable the programmer to see these warnings as they write the code, rather than having to run LPARSE first.

The ability to comment a block of code was also requested. The LPARSE syntax

only supports single line comments from the `%` character to the end of the line (Syrjänen, n.d.a). Therefore commenting and uncommenting large blocks of text requires manually inserting or removing this character from the start of each line. Providing a command to perform this automatically would save the developer time and make this operation more usable.

5.3.2 Launching Feedback

The demonstration of launching LPARSE and SMODELS failed to function on the machine that was used for the demonstration. However, the functionality had been successfully tested on both Windows XP and SUSE Linux systems prior to the demonstration, so pending further investigations into the cause, it was assumed to be a problem related to the Windows 2000 machine on which the demonstration took place. Although this could have potentially limited the results of the demonstration, the launching of a "Hello World" Java application was used as a compromise, in order to demonstrate the Eclipse launching mechanism working.

As part of the demonstration of the JDT, the background compilation of Java source files was demonstrated, and it was suggested that this could also be applied to the grounding of LPARSE source files. However, it was mentioned by the users that large LPARSE programs can take a lot of time, memory and disk space to ground and thus this technique would not be viable. This also underlines the value of having tools to warn the user of potential problems, before the grounding process takes place.

Given the cost of grounding some LPARSE source files, the ability to run LPARSE and save the grounded output to a file was also requested. This would allow a previously grounded file to be reused many times, whilst only incurring the cost of grounding once. In addition, it was also requested to be able to save any output from the console. This feature is already built into Eclipse, as the Common tab on the Launch Configuration dialog allows the user to specify an output file for the launch in addition to the console.

In order to facilitate setting up the LPARSE and SMODELS command line, it was requested that a graphical user interface for selecting these arguments should be available. This would alleviate the need for the user to remember the command line syntax and avoid any errors caused by mistakenly entering the wrong option. However, this ease of use would have the cost of updating the interface when new versions of LPARSE and SMODELS are released.

It was also requested to set the default warning option to `-Wall` as this option to emit all warnings in the file is usually set when running LPARSE. However, if all LPARSE warnings were highlighted in the editor it would not be necessary to run a program through LPARSE in order to see them. The LPARSE warnings should therefore be set on a global, and potentially per project, basis to filter the warnings displayed in the editor, rather than outputting the warnings in the console.

As identified in the requirements (section 3.3.2), the output from SMODELS is often not in a useful form to analyse, as a stable model is represented as list of atoms on a single line (Figure 5.4). In practice the output from SMODELS is often passed to another program (e.g. Perl script) in order to present the output in a more comprehenisble form, e.g. a model of a chess game represented as a grid with symbols for each piece. The IDE should therefore support piping the output from

SMODELS to a command specified by the user.

```
smodels version 2.26. Reading...done
Answer: 1
Stable Model: wants(john,sally) black(sally) sheep(sally) she...
False
Duration 0.061
Number of choice points: 0
Number of wrong choices: 0
Number of atoms: 5
Number of rules: 5
Number of picked atoms: 0
Number of forced atoms: 0
Number of truth assignments: 5
Size of searchspace (removed): 0 (0)
```

Figure 5.4: Sample SMODELS output

5.4 Conclusion

In order to further develop and validate the requirements of the IDE, a prototype system was developed and demonstrated to users within the department. The demonstration session identified additional editor features that could be included in the final system, such as further syntax highlighting and automatic indentation, as well as improvements to the launching mechanism. Again not all of these features could be considered for this dissertation, therefore the following subset was chosen:

- Extending syntax highlighting to constants, `:-` operator and `.` character
- Automatic warning underlining
- Block commenting
- User interface for configuring command line arguments
- Separate launching of LPARSE and SMODELS

The demonstration session was limited due to the launching mechanism failing to function on the demonstration machine. It would therefore have been better to test the system on this machine before the session, in order to check that it functioned correctly and resolve any issues. However, the cause of the issue needed to be investigated in order to avoid it being duplicated in the final system and this is described in section 6.1.4.

Chapter 6

Further Design, Implementation and Evaluation

We have just considered the design and implementation of the prototype system, to validate the requirements obtained in the initial requirements stage and identify additional requirements of the system. As discussed in section 4.1.1, one shortcoming of developing a system using an evolutionary approach is that the structure of the system can be corrupted as development of the system progresses (Sommerville, 2001). Although the initial proof of concept system was discarded and replaced with a higher-fidelity prototype, this system still had to be developed rapidly. This was necessary, in order to reduce the amount of development effort that would have been wasted should the system have been rejected during the demonstration to the users. The rigorous coding standards required for a system in which the users can have confidence, could therefore not be followed whilst developing this system.

In order to develop a more robust system that could actually be delivered to the community, the code was restructured to meet the architecture defined in section 4.2 and the structure of existing features were improved. This was done to facilitate the future maintenance of the system and the reuse of the implemented functionality by a developer wishing to extend the system with additional plug-ins.

In addition to this restructuring, additional features that had been identified in the initial requirements gathering phase and demonstration of the prototype were incorporated into the system:

- More detailed syntax highlighting
- Improved user interface for specifying all LPARSE and SMODELS command line options during launching
- Piping of SMODELS output to another script or program
- Automatic syntax checking
- Display of program dependency graph

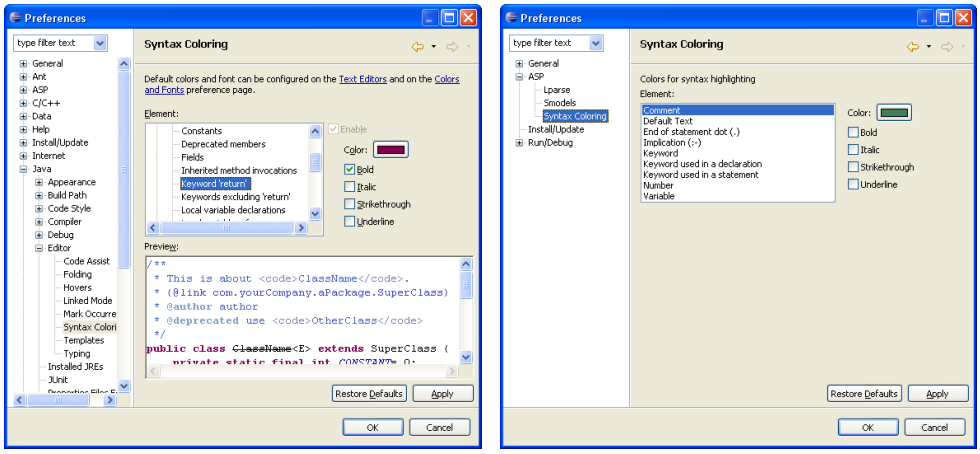
Again, given the exploratory development approach used for this dissertation, this phase also aimed to further develop the requirements specification for the IDE and validate any requirements that had been implemented. This was performed through observations of a user attempting to use the system to solve a small problem using ASP, as each increment of deliverable features was completed.

Let us now consider this further design and implementation of the system in more detail, together with a discussion of the observations made during the evaluation of these increments.

6.1 First Increment

6.1.1 Refactoring of Syntax Highlighting

The syntax highlighting preference dialog implemented for the prototype system (Figure 5.2b) was developed as a subclass of the `FieldEditorPreferencePage` class using `ColorFieldEditor` controls to edit the colours. This approach was chosen as these classes automatically handled the loading, validating and saving of the preferences (Clayberg and Rubel, 2004) reducing the time taken to develop the dialog. In order to reduce this time further, control over other font attributes, such as bold or italic, was not included in the prototype version. When implementing these options in the actual system, the dialog was redesigned to match the JDT for consistency (Figure 6.1). However as the dialog no longer contained only field editors, the more general `PreferencePage` class had to be used, and therefore incurred this additional overhead. The name of the page was also changed from *highlighting* to *coloring* to match the terminology used by the JDT.



(a) JDT Syntax Coloring Dialog (b) ASP Syntax Coloring Dialog

Figure 6.1: Syntax Highlighting Preference Dialogs

As code to handle reading the syntax coloring preferences was now located in both the dialog and the `LparseRuleScanner`, this was abstracted into a set of classes to wrap the low level preference API. This would also allow developers of other plug-ins to the IDE to reuse these syntax coloring preferences without having to know the keys used to access these in the preference file.

The list of elements that could be highlighted was implemented with the `SyntaxColoringElement` class. This class is an enumerated type as "*its legal values consist of a fixed set of constants*" (Bloch, 2001). Given that there is no direct support for enumerated types in Java 1.4, a common pattern is to define a class with a set of string or integer constants representing each valid value in the enumeration.

However this approach is not type safe, as any method that can accept one of the constants as an argument will also accept any other string or integer value, and this can only be checked at runtime.

Bloch therefore advocates the use of the typesafe enum pattern, in which the class contains a set of static constants of the same type as the class and has a private constructor to prevent any further instances from being created. With this approach, any method that accepts the enumerated type can only be passed one of the constant objects defined in the class (or null), otherwise the class will fail to compile. This pattern was therefore adopted for the `SyntaxColoringElement` class.

The `SyntaxColoringPreferences` performed the wrapping of the preferences API, providing static methods to read and write the values of the preference for the individual font attributes of the specified syntax element. In order that the client code could be informed of changes to these preferences, the `SyntaxColoringPreferenceListener` interface was defined which contained callbacks for updates to each of the font attributes. Client code could then implement a class to handle the changes and register this with the `SyntaxColoringPreferences` class in order to receive notifications.

It was envisaged that in the future, the system may need to support syntax highlighting configurations that were not stored in the workspace preferences. An example for this would be contributing a predefined syntax highlighting scheme to the IDE, such as the emacs highlighting scheme used in the SMODELS mode. In order that the `LparseRuleScanner` could work with other types of configurations the `ISyntaxColoringConfiguration` interface was defined for accessing the configuration properties, and the `SyntaxColoringConfigurationListener` interface for listening to changes to the configuration. The `SyntaxColoringConfiguration` class was implemented to provide a default implementation of this interface by delegating to the `SyntaxColoringPreferences` class. The scanner would therefore use the API defined by the interface rather than the concrete class.

In addition to this refactoring, rules for recognising the end of line symbol `.` and the `:-` operator were added to the scanner class as requested in the demonstration of the prototype (section 5.3.1). However, as was observed in section 5.2.1, the highlighting of other tokens such as constants and functions could not be achieved without additional parsing of the source file.

6.1.2 Parsing of Source Files

In section 5.2.1 we discussed the process of lexical analysis in which the source program is broken up into a sequence of tokens. The next phase in the analysis of the program source is called syntax analysis, and is implemented by a program called a parser (Wilhelm and Maurer, 1995). The parser attempts to recognise the syntactic structure of the program by matching the lexical token sequence against a specification of the programming language, known as a grammar. In addition to this, it builds a data structure representing this syntactic structure which can then be used by the later analysis stages.

Thus parsing of the LPARSE source files is not only necessary for the more detailed syntax highlighting, but for any other tools that need to perform an analysis of the program. This includes highlighting of errors and warnings in the editor, com-

putation of dependency graphs, autocompletion and analysis of program tightness. Given the need for many tools to perform this process, the data integration approach described in section 2.4.1 was adopted, in which the source code is only parsed once and stored in a shared data structure that could be used by several tools. We consider this data structure in the second increment (section 6.2.2), but for the first increment concentrate on the implementation of the recognition phase.

In order to parse the LPARSE source files a definition of the language's grammar was required. Although a description of the grammar was available in the technical report by Syrjänen (1998), this was outdated and did not reflect the current version of LPARSE. The only current documentation available was the LPARSE source code itself. The LPARSE parser is written as a specification for the parser generator tool, Yacc, and the corresponding lexer as a specification for the lexical analyzer generator tool, Lex. Both these tools transform their specifications into C programs that perform the corresponding analysis phase (Appel and Ginsburg, 2004).

Given that part of the Yacc specification is the set of grammar rules defining the language, the required grammar could easily have been extracted from the source code and consequently used to develop a new Java-based parser for the IDE. However given that LPARSE is licensed under the GNU GPL, modification of the source files and the inclusion of these files in another work is permitted as long as the GPL is also applied to that work (Free Software Foundation, 1992). Given the time constraints of the dissertation, it was therefore decided to modify the parser and scanner specifications to work with Java and reuse them in the IDE, as this would avoid the cost of redeveloping the parser from scratch. Using the same specification would also ensure that the IDE's parser accepts the same programs as the LPARSE tool itself.

As the Lex and Yacc tools output C programs, they could not be used directly by Eclipse to build a Java data structure. Although, it would have been possible to interface with these programs using the Java Native Interface (JNI), this would have introduced the overhead of learning to use this approach, as well as the introduction of issues such as potential memory corruption and the high cost of moving between Java and native code (Bloch, 2001). This would also have led to a dependence on the underlying platform, requiring these components of the IDE to be built for each platform on which it would need to run, rather than having a single distribution. However, there are ports of both Lex and Yacc for Java, so it was deemed to be easier to modify the existing files to be compatible with these tools.

The scanner specification was converted to work with the JFlex[1] tool. This was achieved by removing any C code from the file, updating some of the tool directives, and replacing the action code to generate tokens that would be understood by the Java parser generator. JFlex is designed to work with both the CUP[2] and BYacc/J[3] parser generators. The latter tool was chosen for use in the IDE, as it required fewer changes than CUP which used a slightly different syntax to the existing Yacc specification. As only recognition of the program was to be performed in this increment all of the action code used to build the existing C data structure was removed from the specification.

[1] http://jflex.de/
[2] http://www2.cs.tum.edu/projects/cup/
[3] http://byaccj.sourceforge.net/

The scanner used in the LPARSE tool performed the analysis of a single program combined from all the input files specified on the command line. However for a file open in the LPARSE editor, it is not known with which other files it would be used or whether it would even be used with any other files. Therefore when programs are split over multiple files, some way of defining how these files are aggregated to form a single program would be necessary to perform an analysis of the entire program. However, there was insufficient time available in this dissertation to perform a full consideration of how to achieve this. By limiting the analysis to a single file, the file currently open in the editor, the features built on top of this parser could still be developed and validated by the users. Therefore in the IDE the scanner was initialised with a single `IDocument` object used by the text editor to represent the document being edited.

In order that the future data structure modelling the program could be updated as the user types, and thus allow any of the features depending on the parser to work with an up to date representation of the program, the parsing process was incorporated into a reconciling strategy. A reconciler is used to update the underlying object model of the program represented by the editor as changes to the contents of the editor are made (Arthorne and Laffra, 2004). Any changes made to the file being edited are placed in a queue and processed by the reconciler to update the model. The reconciler used in this project was the `MonoReconciler`, as this runs in a low priority background thread and waits for several modifications to be queued before reconciling. This allows the user to continue editing the document while the reconciling is taking place.

6.1.3 Highlighting of Syntax Errors

A syntax error occurs "*when the string of input tokens is not a sentence in the language*" (Appel and Ginsburg, 2004). In order that as many syntax errors as possible can be reported to the programmer in a single pass through the program, the parser should be able to recover from the discovery of an error and continue to discover other potential errors (Wilhelm and Maurer, 1995). A simple error recovery mechanism, described by Appel and Ginsburg (2004) is to skip over tokens until one is reached from which the parsing can resume. An LPARSE program consists of rules, statements and declarations each separated by a `.` (Syrjänen, n.d.a). Therefore after encountering an error the parser can skip over the tokens until this token is encountered, knowing that the following token sequence should correctly match a rule, statement or declaration providing that no further errors are encountered. Indeed this is the method of recovery used in the original parser, and has been maintained for the IDE.

In the generated parser, the `yyerror` method is called whenever a syntax error is encountered. In order to highlight these errors in the editor, the `LparseSourceProblem` class was defined to provide a data type for storing the location of the error in the source file together with its description. As the source file is parsed and errors are encountered, an instance of this class describing each error is created and placed in a list.

In Eclipse, marker objects can be used to attach annotations to workspace resources, with these annotations being stored in the workspace metadata rather than by mod-

ifying the existing file (Clayberg and Rubel, 2004). The Eclipse text editor automatically highlights errors and warnings in a file, if problem markers representing them are attached to the resource (Figure 6.2). These are also displayed in the Eclipse problems view. Therefore following the parsing of a source file, any existing problem markers are removed from it and problem markers are added for each `LparseSourceProblem` in the list produced by the parser.

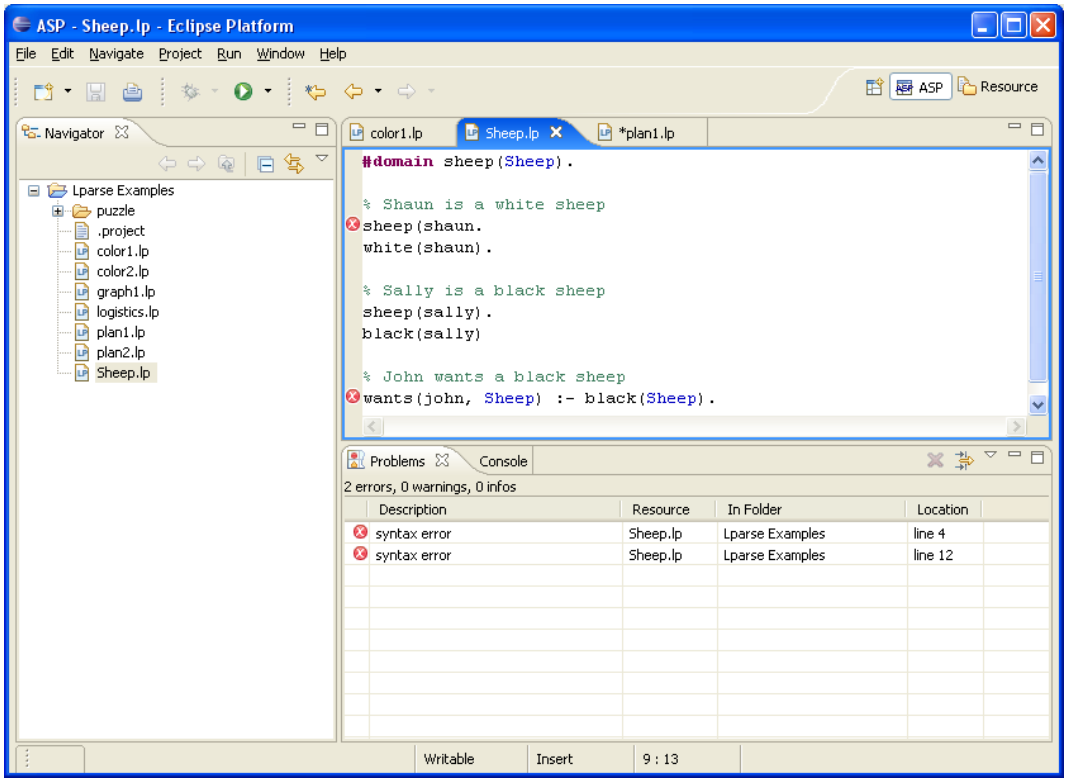


Figure 6.2: Highlighting of Syntax Errors

In order to highlight warnings and other errors in the program, more than just an analysis of the syntax of the program is required. We therefore consider the extension of the error highlighting to encompass these, following the implementation of the data structure representing the program (section 6.2.3).

6.1.4 Observation Session

Following the completion of this first increment, an observation of the usage of the system was conducted in order to further develop and validate the requirements of the system. According to Sommerville (2001), one of the fundamental characteristics of evolutionary development approaches is the involvement of the stakeholders in "*designing and evaluating each increment*". It was thus important to make the system available to the users at this point in the development of the system, in order that these aspects could be discussed with them.

The observation undertaken for this evaluation combined aspects of both the usability testing approach, discussed in section 2.5.3, and ethnography. Indeed, Sommerville observes that combining the latter technique with prototyping can help

reduce the number of refinement cycles that are required. More importantly, in ethnography the analyst immerses themselves in the environment in which the system is to be used in order to gain a better understanding of the processes in which the user is involved. As we have previously considered, an understanding of the programming practices used in ASP would help to identify the areas which require better support or possible improvements to these practices. This approach was therefore chosen in order that this aspect could be considered as part of this dissertation

Given the timescale of this dissertation and its general focus on the elicitation of requirements of the IDE, the consideration of the usability of the system was not intended to be an in depth study, rather a means of identifying any fundamental usability issues present in the system. Additionally, in order to combine this with the ethnographical study considered above, the evaluation session had to be carried out in the field. This meant that the support for more sophisticated data capturing techniques, such as screen capture and keystroke logging, that would have been available in a laboratory could not be used. Thus the evaluation undertaken was more of a 'quick and dirty' approach (Preece et al., 2002), rather than a formal usability test.

As we have already observed, the resources of this dissertation have limited any direct contact with users of ASP to those within the department. Following the specification of a non-trivial problem that they intended to implement with ASP, one of the users within the department contacted the author offering to use the IDE to tackle the problem and be observed doing so. Therefore, regarding the ethical concerns of observational approaches highlighted in section 2.5.3, the consent of the participant to participate in the evaluation was clearly provided. After further discussions with the participant it was decided that for the initial evaluation session, this task would be substituted with one that was deemed to be simpler - encoding a Rubik's cube as an ASP program. It was felt that this would reduce the amount of time spent designing the system, and allow more of the session to be focussed on system usage. The session was conducted at the subject's Linux workstation and lasted approximately three hours. Let us now consider some of the observations made during the evaluation session.

Observations

The first observation that was made during the session was that upon opening the editor, the subject entered a header containing the file name, author, email address, date and a description of the program. This served as a form of documentation of the purpose of the file for future reference. This technique was also observed in some of the example programs provided with the LPARSE distribution. As this activity is always performed when a new file is created, it would be useful to provide a template for new source files prepopulated with known fields such as the filename, author and email. This would reduce the overhead of performing this each time, and thus potentially encourage other users to adopt this methodology. Indeed, the JDT provides a similar template for new Java files demonstrating that this would be feasible to achieve in Eclipse.

A usability problem that was encountered was that the cut, copy and paste menu items were disabled on the main Eclipse menu bar, although they were present on

the editor's right-click menu and through keyboard shortcuts. This would disadvantage users whose interaction style relies heavily on the use of menus, rather than keyboard shortcuts, and thus falsely imply that this functionality was not available in the editor. This defect was corrected in the second increment, by setting the contributorClass attribute of the editor extension in plugin.xml to the `TextEditorActionContributor`. This allowed the LPARSE editor to contribute the standard Eclipse text editor commands to the menu bar.

When writing the program the subject only used single letter variable names, such as `X` or `T`. This lead to the subject scanning through the code to determine which variables they had already used, for example commenting "*Have I used T?*". This process could be facilitated by providing a variables view for the editor, displaying all of the variables currently present in the editor. This view could also be combined with the domain of the variable, for variables defined with a domain declaration (Syrjänen, n.d.a).

However, given that the LPARSE syntax does not impose such a limit on the length of variable names, the programmer could instead have used a more descriptive name. This would make the domain of the variable clear from its name, rather than through an additional view incorporated into the IDE. This is arguably an area where better programming practices could be employed rather than additional tools, as adopting better naming conventions would be beneficial to those reading the code without the support of the IDE. It may be argued that shorter names reduce the amount of typing required, and consequently the number of errors made when inputting the program. However incorporating autocompletion mechanisms instead could help support this aspect of the programming process, whilst maintaining the readability of the raw code.

Another issue with the usability of the system that was noted was the font used by default by the editor under Linux. The size of the font was too small for the subject to read, especially the full stop at the end of the line, which was hardly visible even with the additional syntax highlighting. This was not the case under Windows, where the Courier font was used. Although the Eclipse platform does provide a dialog for configuring the font used by editors, it was suggested that a better scheme should be provided by default for the LPARSE editor.

The program written by the subject was divided into two separate files, one to model the Rubik's cube problem and the other to define a test scenario. This separation allowed the main definition of the problem to be potentially reused with multiple scenarios. Given that this methodology is used in practice, it validated the requirement for the IDE to support programs over multiple files.

In the demonstration of the prototype system the launching mechanism failed to function (section 5.3.2). This problem was not resolved prior to this observation session as it could not be reproduced when tested on other Windows and Linux systems, making it difficult to identify the cause of the problem. However it was reproduced during the observation session on the Linux system used by the subject. After checking the Eclipse logs and the version of Java on the subject's system, it transpired that this was due to the use of the Java 5.0 `ProcessBuilder` class, which could not be loaded on either machine as an older version of Java was present. It was therefore decided to remove the use of this API from the system and replace it with equivalent code from the Java 1.4 API in order to make the IDE compatible

with a larger number of environments. The launch configuration used to test the plug-in on the development machine was also modified to launch the IDE using this older version of the Java Runtime Environment (JRE).

The presence of this issue meant that the launching element of the IDE could not be used during the evaluation session, causing the user to resort to using the command line. However this did serve to illustrate how switching between the editor and the command line interrupted the flow of the programming process and thus validated the requirement to include them in the same environment. Moreover, a Perl script was written to present the raw output from SMODELS as a series of grids representing the state of each face of the cube. This was required because the information was otherwise difficult to extract from this raw output. As support for piping to another process had not yet been implemented in the IDE, the subject would still have had to resort to using the command line. This therefore also validated the need for this feature, and it was decided to incorporate this into the next increment.

The requirement for block commenting functionality that was identified in the demonstration of the prototype was again requested by the subject. In order to observe whether a set of rules was having an effect on the output of the program these were commented out. However, this was time consuming to perform as each line had to be commented out individually. An alternative to providing this support in the IDE would be to introduce multi-line comments into the LPARSE syntax, as is available in languages such as C and Java. This would support this process without obliging the user to use the IDE.

Following the session the subject commented that they would have preferred to use Emacs shortcuts, rather than those provided by default in Eclipse. However, the Eclipse platform provides the ability to configure the shortcuts used for each command and groups these configurations into a scheme. It also provides an Emacs shortcut scheme by default (Figure 6.3). This functionality would enable users to adapt more quickly to using the IDE, as they could use the shortcuts that they are familiar with, rather than having to learn the commands provided.

6.2 Second Increment

6.2.1 Improvements to Launching Mechanism

During the demonstration of the prototype system it was requested that the output from LPARSE should be able to be saved by the user, in order to avoid grounding a program each time that it needed to be run through the solver (section 5.3.2). In order to achieve this, a separate launch configuration type and tab group were created for LPARSE, allowing it to be run by itself in addition to with SMODELS.

In order that a new LPARSE process could still be started by SMODELS, the `LparseLaunchConfigurationDelegate` class provided an additional method to just start the process rather than launching it within the Eclipse launching framework. Similarly the launch configuration tabs for LPARSE were made public API in order that the same user interface for configuring LPARSE could be used from the SMODELS tab group. This also allows other developers to reuse this functionality, when writing plugins for other tools that use LPARSE as a front end.

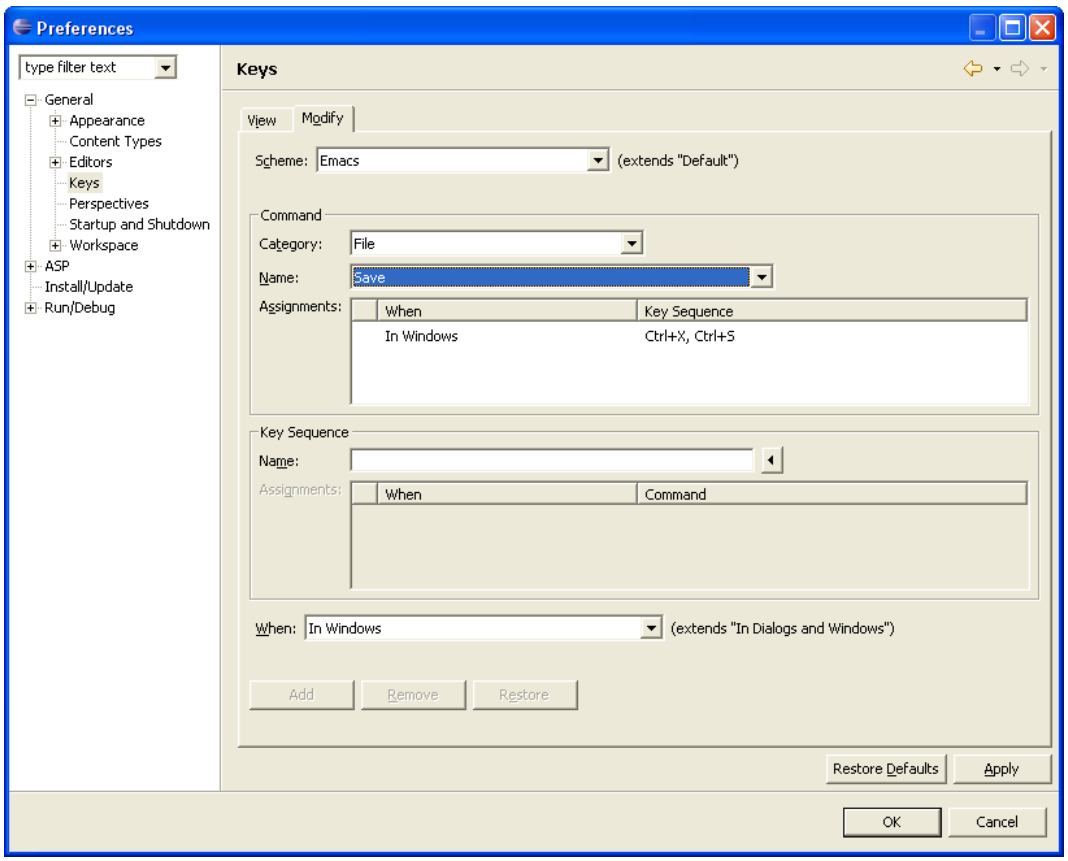


Figure 6.3: Configuration of Emacs Keys

As was also mentioned in the demonstration, the configuration dialog needed to provide a user interface for controlling each command line option rather than simply providing a text field in which to enter the arguments. This requirement was met by implementing separate tabs to specify the input files (including previously grounded files), constants, and the other LPARSE and SMODELS arguments as defined in Syrjänen's user manual. The options were labelled using the plain text description in the user manual, but this was supplemented with the name of the argument to support users familiar with the argument name (Figure 6.4). The launch configuration delegate classes were then updated to build the command line from these new specific options.

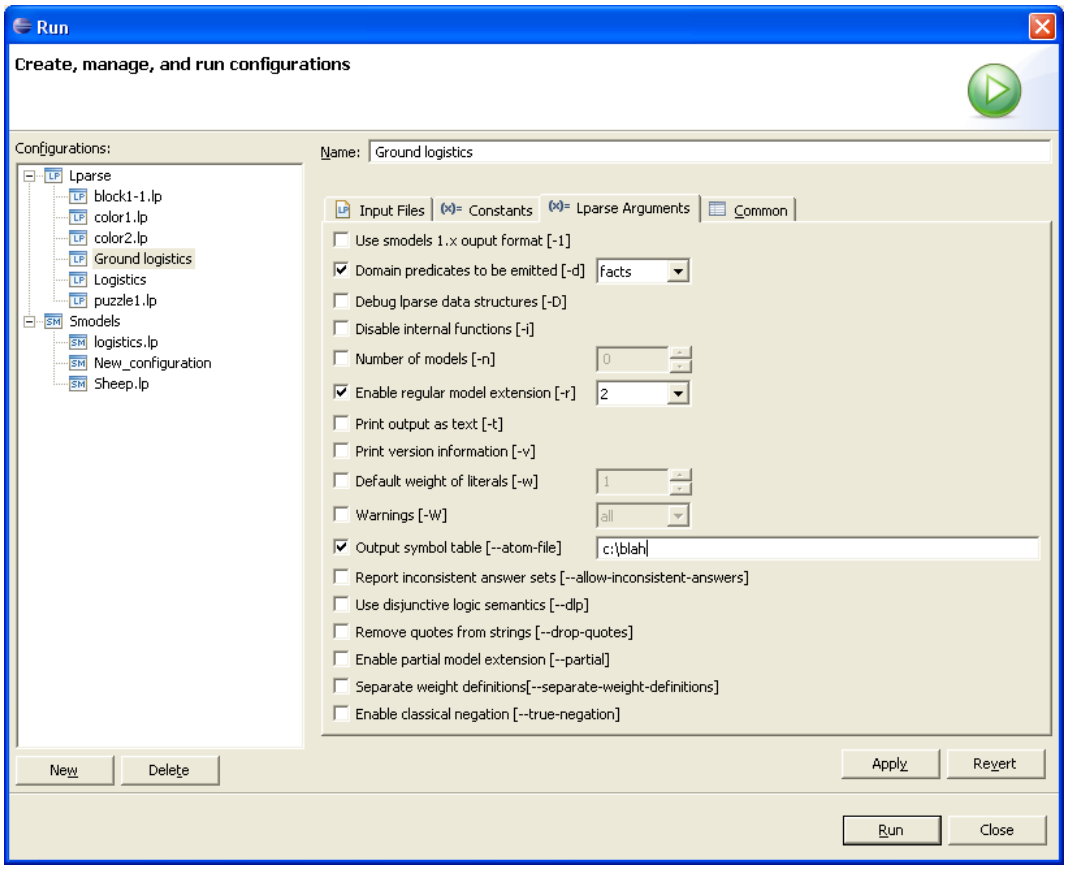


Figure 6.4: LPARSE Arguments Tab

Following the observation session, it was identified that the requirement for piping the SMODELS output to another program had to be implemented before the following session, in order for the subject to be able to use the IDE without switching between the IDE and command line. An additional tab was therefore created with the option to output directly from SMODELS or through another command specified in the text field (Figure 6.5). The `SmodelsLaunchConfigurationDelegate` class was then modified to launch this external process, if specified, and pipe the output from SMODELS to it.

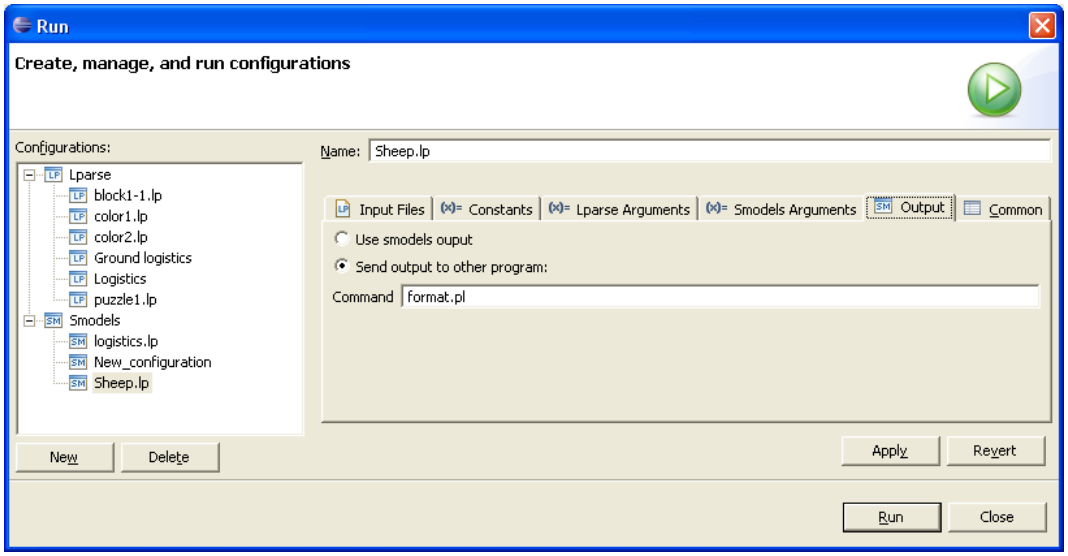


Figure 6.5: SMODELS Output Tab

6.2.2 Shared Data Structure

In section 6.1.2 we considered how a data structure representing a model of the source program was required to implement further features in the IDE. In a compiler, the parser generates a tree structure representing the syntax of the program, which can then be used in the semantic analysis phase (Wilhelm and Maurer, 1995). This structure may represent the concrete syntax of the program, as defined by the grammar of the language, or discard unnecessary elements that are not required for further analysis of the program, such as keywords, forming an abstract syntax. Other elements of the input program are discarded at the lexical analysis phase, such as comments and whitespace, as they are of no use to the compiler (Appel and Ginsburg, 2004).

Although producing an abstract syntax tree and discarding some information would be sufficient for tools that only have to analyse the meaning of the program, such as for drawing a dependency graph, other tools that need to analyse and manipulate the structure of the source code itself, such as refactoring, would need a model of the source file itself. Therefore the data structure was designed to represent the full syntactic structure of the source file, and in addition relate each token back to its location in the file. Before considering this design, let us examine the structure of a grammar in more detail.

The context-free grammars used to define the syntax of a programming language, consist of a set of production rules of the form $symbol \rightarrow symbol\ symbol \ldots symbol$, where a terminal symbol is a token and can only appear on the right-hand side of the rule (Appel and Ginsburg, 2004). Non-terminal symbols can also appear on the left hand side, and a special case of these is the start symbol, representing the whole program. For each of these rules Yacc provides a space for a semantic action, some code that is executed when the associated rule is matched. This is where the code to build the data structure is written.

In order to reflect this hierarchical structure, the natural data structure used to

represent the source file was a tree, whose nodes were represented by the abstract `ASTNode` class (Figure 6.6). This defined methods for accessing the starting position and length, in the original source file, of the section of source code represented by the node, together with the actual source code text represented. It also provided a method for navigating the structure, with a reference to the node's parent.

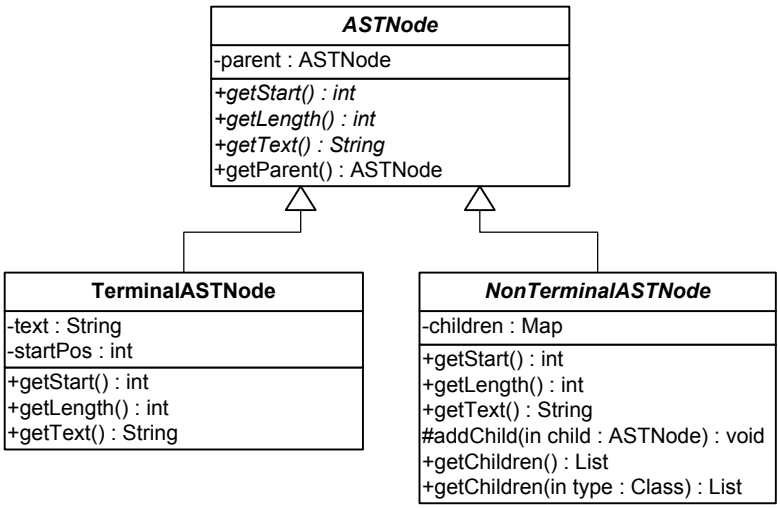


Figure 6.6: ASTNode Class Hierarchy

Subclasses of the `ASTNode` class were defined in order to represent the terminal and nonterminal nodes in the tree. These were further subclassed to represent the different terminal and non-terminal nodes in the grammar, such as rule, variable, atom, statement, constant etc. These could then be embellished with specific methods for navigating the tree, such as the `getHead()` and `getTail()` methods of the `Rule` class. Although the classes defining these nodes were implemented, it was not possible to create the embellishments for all of the classes within the timeframe of the dissertation. Therefore only those that would be used by other features were implemented.

The `NonTerminalASTNode` class was designed to store its children indexed by type. This would allow children of a given type to be accessed without having to search through the list of child nodes each time. This was implemented with a map of the class name to a list of child objects. The more specific subclasses would store the children in this structure defined by this base class, with the embellished accessor methods delegating to the accessor methods of the base class. As terminal nodes could only appear as leaf nodes in the tree, the `TerminalASTNode` class did not define any children, only default implementations of the abstract methods from `ASTNode`.

The `LparseSourceFile` subclass of `NonTerminalASTNode`, representing the start symbol, was embellished with an accessor method for returning a list of rules in the source file, but also other elements that were not defined in the grammar such as comments. These were added to the source file node as they were required to maintain a full model of the source file document, but could not be attributed to any other specific node in the tree as they are not part of the grammar. The

list of `LparseSourceProblem` objects used to describe errors in the code was also stored in this root node, as these errors were also a property of the file as a whole. However in order to attribute the error to the specific node which caused it, the `LparseSourceProblem` class was refactored to be associated with this node rather than just the character in the file on which the error was encountered.

Another additional property in the `LparseSourceFile` was a set of symbol tables, which Appel and Ginsburg (2004) define as "*mapping identifiers to their types and locations*". These tables are used in the semantic analysis of a program to perform type checking, but could also be used within the IDE to highlight identifiers according to their type (section 6.3.1) or highlight all occurrences of a given variable in the source file.

The table was implemented with the `SymbolTable` class, mapping the textual name of each identifier to a list of nodes for each occurrence of the identifier in the source file. A symbol table was defined for variables and each identifier type (symbolic constants, numeric constants, functions, symbolic functions and atoms). As the program was parsed, the identifiers and variables were inserted into the appropriate symbol table once their type had been deduced.

6.2.3 Additional Error and Warning Highlighting

The original Yacc grammar contained code in the action for the constant declaration rule to warn the user if the constant that they were declaring had already been defined or used as a symbolic constant. This was not possible to implement in the first increment of the IDE, as the symbol tables that we have just considered had not yet been implemented. In order to support the warnings in addition to errors, a problem type field was added to the `LparseSourceProblem` class, in order that the type of the problem marker could be set correctly during the reconciliation.

The original grammar also contained rules for common mistakes made when entering constant declarations: using a variable name rather than identifier for the constant or missing the assignment operator. This provided a more specific error message than the general 'parse error' message would have, aiding the programmer to locate the problem more quickly. The action for these rules was therefore implemented to create a new problem object with the same message as provided in the original C code. This could be extended by investigating other common errors made when writing LPARSE programs, adding rules to support them, and returning a problem object with a more specific error message.

As no additional analysis of the data structure was performed after building it, the errors and warnings highlighted in the editor were limited to those we have already discussed. Further improvements to this feature would include implementing the detection of all errors and warnings detected by LPARSE as well as investigating further warnings that may be helpful to the programmer. However, these extensions are not considered in this dissertation.

6.2.4 Observation Session

Following the development of the second increment, a second observation session was conducted. The session aimed to continue to evaluate the system, whilst the

subject attempted to complete the encoding of the Rubik's cube problem using ASP. However it became apparent that the problem needed to be formulated in a different way, and thus most of the session was spent trying to find the best way in which to do this. Given that the problem was harder to encode than previously thought and the formulation of the problem was not providing any input into improvements to the system, the session was cut short, with the intention of attempting a different problem after the completion of the next increment.

Despite this, the system was still used at the beginning of the session from which some feedback, which we will now consider, was obtained.

Observations

The main outcome from the session was that the subject was able to edit and launch the program from within the IDE following the move to the Java 1.4 API and the improvements to the launch configuration dialog. One point that was raised was that not all of the options on the LPARSE configuration tab were visible on the participant's system. In order to redesign the layout of this dialog the non-functional requirement of minimum screen size to be supported by the system would need to be investigated. However, there was insufficient time to do this for this dissertation.

Given that the output of each answer set from SMODELS is output on a single line, the subject had to scroll horizontally in order to analyse this data. Although this was no longer an issue when the Perl script was used to filter the output, it was requested that the console should be able to provide a text wrapping feature. This would prevent the user from having to scroll through the output when just the raw output is used. Upon further investigation this is supported in Eclipse, via the 'fixed width console' setting in the console preferences. However given that this option would be required by users of the IDE, a reference to this in any system documentation would make this option more apparent to the user.

Finally, the block comment functionality was once again mentioned during the period of system usage. It was therefore decided to implement this in the next increment.

6.3 Third Increment

6.3.1 Semantic Highlighting

With the implementation of the symbol tables discussed in section 6.2.2, the type of each identifier in the source file was now known. This enabled an element for constants to be added to the syntax coloring configuration as requested in the demonstration of the prototype (Figure 6.7), together with elements for the other identifiers in order to determine whether these would also be of use to the user.

A `modelUpdated` callback method was added to the `LparseRuleScanner` class in order that it could be informed of any changes to the model by the editor. After receiving the updated model, the rules used in the scanner are updated by iterating over the various symbol tables and adding rules to associate the identifier names with the correct token type.

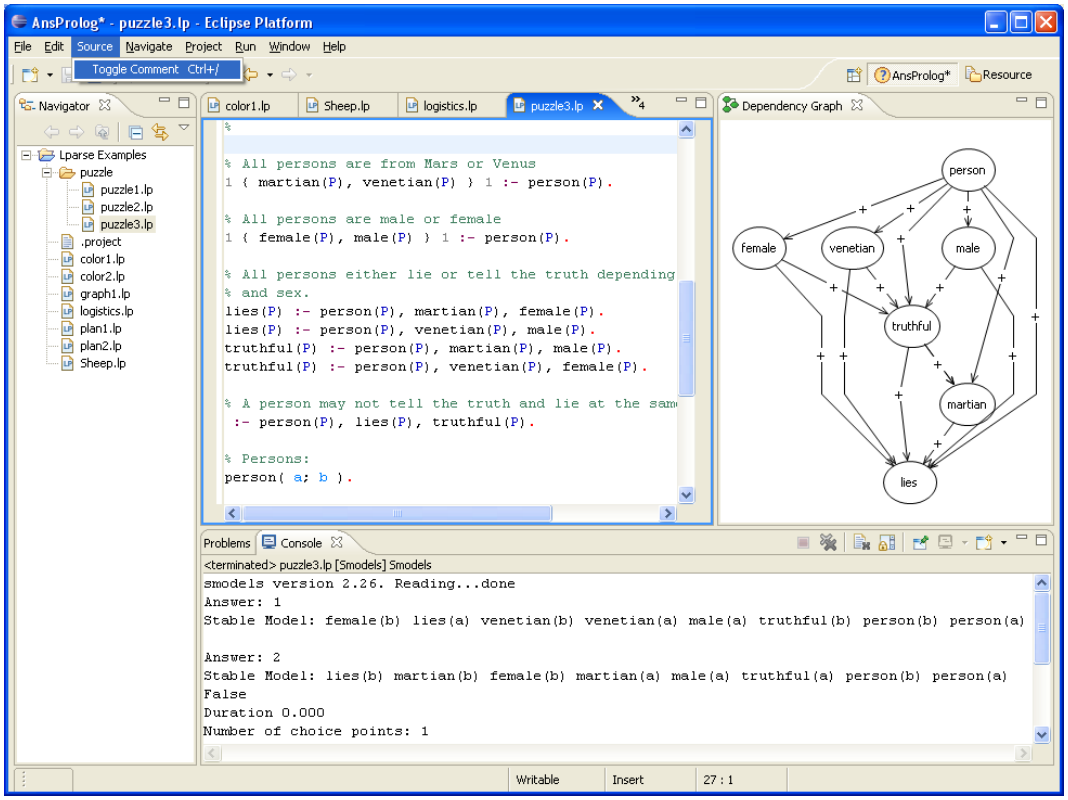


Figure 6.7: Third Increment

6.3.2 Block Commenting

As the block commenting feature had been requested in the demonstration and both observation sessions, it was implemented in this increment. The action was named 'Toggle Comment' in order to be consistent with the JDT, and was implemented to behave in the same way. The action itself was implemented with the `ToggleCommentAction` class, a subclass of the `Action` class, overriding the `run()` method to implement the toggling of comments on the lines of code currently selected in the editor.

In order to contribute this action to the menu bar (Figure 6.7), a subclass of `TextEditorActionContributor` was implemented in order to keep the existing menu items such as cut, copy and paste (section 6.1.4). The corresponding attribute in plugin.xml was also updated accordingly. In this contributor, a top level *Source* menu was created in which the action was placed, again to be consistent with the location of this command in the JDT.

Entries were also added to plug-in.xml in order to specify the keyboard shortcut used to activate this command, which was again set to be the same as used in the JDT: `Ctrl + /` under the default configuration and `Ctrl + 7` under the emacs configuration. However the use of `Ctrl + %` for the default configuration may have been more natural for the user, given that `%` is the single line comment character for LPARSE, rather than the `//` used in Java. It was felt that it would be better to use the same command, in order to prevent users of both plug-ins having to learn different commands for each language within the same IDE. In any case, the key

settings could be reconfigured by the user if required (section 6.1.4).

6.3.3 Dependency Graphs

The questionnaire demonstrated strong support for the display of dependency graphs (section 3.3.1), therefore this feature was chosen to be implemented in this increment given the availability of the source file model. Baral (2003) defines the dependency graph of a program to consist of:

- a set of vertices, such that each vertex corresponds to a predicate name.
- a set of edges, such that the edge from P_i to P_j is in the set if and only if there exists a rule in the program that has P_i in the head and P_j in the body. The edge is labelled with a + if P_j appears as a positive literal, with a - if it appears as a negative literal, or indeed with + and - if rules exist such that both cases are present.

The LPARSE syntax supports special rule types called choice, constraint and weight rules, in addition to its other elements such as declarations and statements (Syrjänen, n.d.a). However, the definition of a dependency graph is only concerned with rules, whether a literal is present in the head or tail of the rule, and whether that literal is positive or negative. In addition, the dependency graph functionality was defined in a separate plug-in (section 4.2), in order that it could be reused by plug-ins for other ASP tools. As each of these would potentially have their own special syntax and source file model, an intermediate data structure was designed as input to the plug-in, simplified to contain only the elements required to compute the dependency graph (Figure 6.8). The `DependencyGraphBuilder` utility class, in the LPARSE UI plug-in, was written to implement the translation from the `LparseSourceFile` model to the `AnsPrologProgram` data structure.

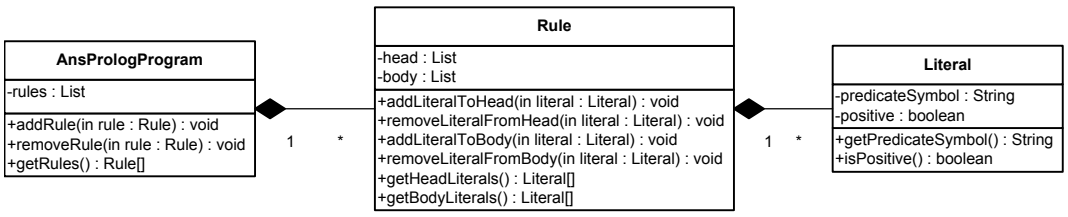


Figure 6.8: Intermediate Data Structure for Dependency Graphs

In order to display the dependency graph in Eclipse a suitable library to support graph drawing had to be chosen. Although this could have been written from scratch, it was deemed that implementing other features and investigating these would be a better use of the limited time available than re-implementing existing functionality. Commercial packages were not considered due to the cost and licensing issues.

One potential package that was suggested by the evaluation participant was the open source GraphViz[4] software package which provided tools to support the layout and display of graphs. The package used a simple text based syntax for defining graphs,

[4] http://www.graphviz.org/

a list of edges of the form `node1->node2`, so once the graph had been computed from the abstract representation of the program it would have been easy to write out the list of edges in this form. However as the tools were not Java based, the user would require a copy of this program built for their specific machine, which could be launched from Eclipse in a similar way to LPARSE and SMODELS. However as this would display the graph in the viewer application's own window, rather than as a view in Eclipse, the integration of this component into the IDE would not have been achieved. In addition, the user would have to launch the graph viewer each time they wanted to see an updated graph, rather than having this information automatically displayed within Eclipse.

The Grappa[5] package is a Java based subset of GraphViz for building and displaying graphs. However this framework uses Sun's Java2D and Swing libraries for displaying graphs, rather than Eclipse's SWT. Although SWT does provide a compatibility layer with the `SWT_AWT` class, this is only compatible with Java 1.5 and Windows, Linux/GTK and Linux/Motif (Eclipse, 2005d), and would therefore limit the portability of the IDE. In addition, the library does not perform any layout of graphs, so the user would still require a build of the GraphViz layout engine on their machine.

In order to provide an SWT based graph viewer, the chosen solution was to use the Draw2D plug-in from the Eclipse Graphical Editing Framework (GEF) feature. The Draw2D plug-in provides support for drawing figures on an SWT canvas (Lee, 2003), and contains a graph package containing classes for modelling and laying out graphs. However it did not provide direct support for drawing the graphs, allowing the client code to define the desired look of their graph by using the computed layout information to build their own drawing from the shapes provided by the library. The `GraphViewer` class was therefore implemented to wrap a Draw2D `FigureCanvas` control that would draw the supplied graph.

In order to display the graph within the IDE, the `DependencyGraphView` was implemented, which given an `AnsPrologProgram` data model would build a dependency graph model using the Draw2D graph classes and display this in its `GraphViewer` (Figure 6.7). The LPARSE editor was also updated to call the view's `displayDependencyGraph` method after its model is reconciled, in order that the graph displayed in the view reflects the current state of the source file open in the editor.

However, there were some problems with implementing the graph view using the Draw2D package. A precondition of the `DirectedGraphLayout` is that the graph has to be connected, however this is not necessarily the case for the graph of an ASP program under development. Thus dummy edges were added to the graph to ensure that the graph was connected before passing it to the layout class, and these were removed prior to displaying the graph. The graph could also not be laid out if a vertex had a self loop indicating that the predicate depended on its own truth or falsity. Therefore these edges were removed from the graph and could not be displayed. In order to provide a complete implementation, a better layout algorithm would need to be implemented or an alternative approach adopted, but there was insufficient time available to consider this. However, it is was felt that the dependency graph feature implemented would still be a valuable tool for demonstrating this concept and further investigating its potential.

[5] http://www.research.att.com/ john/Grappa/

6.3.4 Observation Session

Although it was originally intended that a further evaluation session would be performed after the delivery of this increment, there was insufficient time remaining to complete this. However, this process of implementation and evaluation would ideally have been continued in order to further develop the IDE and its requirements.

6.4 Conclusion

The interleaved process of implementing the system and evaluating it with the users, has achieved its aim of producing a working system and further developing the requirements. Although the system produced did not implement all of the features defined in the requirements due to the time constraints of the dissertation, and some features such as the dependency graph view still had some unresolved issues, the system could still be released to the users subject to some further work and testing (discussed in the next chapter).

The observation sessions served not only to identify further requirements such as templates for new files, but to demonstrate the importance of some of the existing requirements, namely block commenting and piping SMODELS output to an external program. The use of the system to actually perform an ASP programming task allowed any usability issues, such as the disabled editor functions, to be corrected for the next increment of the system. Furthermore, the observation of the user's programming practices allowed improvements to the way in which variables are named to be suggested as an alternative to new tools.

Given that the final increment of the system could not be evaluated within the time frame of the dissertation, there is scope for this process to be continued in the future in order to further develop the requirements of the system and its functionality.

Chapter 7

Testing

The exploratory development approach that was adopted aimed to investigate both the requirements of the IDE and to develop the system. Therefore, there was a tension between rapidly developing as many features as possible in order to use the system to get a better understanding of the requirements, or developing a smaller subset of features and testing these thoroughly to produce a robust system that could be used by the community. It was decided that this dissertation would focus on the former approach, so let us now consider how the test plan was designed to fit this.

7.1 Test Plan

According to Stiller and LeBlanc (2002), testing the individual system components is the first stage of the testing process. A set of test cases are written to demonstrate whether each component functions correctly. This is known as unit testing. After checking that each component works correctly, integration testing should then be performed to verify that there are no problems with the interactions between the individual modules.

Given that the Eclipse plug-ins are developed in the object-oriented language, Java, the plug-ins written for the IDE are composed of a set of classes. Therefore unit tests could be written to test the behaviour of each class, followed by testing the functionality of the system as a whole. Indeed in their book *Building Commercial Quality Plug-ins*, Clayberg and Rubel (2004) describe the JUnit testing support provided by Eclipse for testing Java classes and the support provided by the PDE to extend this to testing the functionality of plug-ins. Indeed they advocate developing a suite of automated tests in order to facilitate testing plug-ins for regressions as the system is developed. This would also facilitate multi-platform testing, as the test suite would only need to be written once and could then be run on all of the target platforms.

However as it was decided not to develop a smaller 'commercial quality' system, but a larger exploratory system, this approach was not adopted for this dissertation. The time taken to develop a good set of unit tests to cover both normal and extreme inputs to each of the classes would have taken a considerable amount of time, and this was therefore not suited to the rapid development approach adopted. However

it was still important to perform some testing of the system in order to avoid any errors affecting the demonstration or evaluations of the system.

Since the only usage of the system within the dissertation would be in these sessions, it was considered sufficient to check that each feature would appear to work correctly to the user. That is to say perform manual testing of each feature through normal usage, to test that each feature appears to operate as expected and does not generate any unexpected exceptions. For example, if the generated dependency graph occasionally contained an incorrect vertex or edge this would be unlikely to be detected by the user, and would therefore not disrupt the evaluation process. However, if the block commenting menu item did nothing or an error message was displayed when opening the editor, then these features could not be demonstrated or evaluated by the user. It was also considered reasonable not to spend time stress testing the system or verifying how it would respond to extreme data input, as it is unlikely that this would be reflected in its usage within the demonstration or evaluations.

However if the system were to be delivered to the users in the future, the detailed testing approach that we have just considered would need to be applied to ensure that the whole of the system does indeed function correctly. For example, it would no longer be acceptable for the dependency graph view to display incorrect results, as this would mislead the programmer using the system.

Before considering some examples of the testing that was undertaken, let us first examine the test environment that was used.

7.1.1 Test Environment

The Plug-in Development Environment (PDE) provides a runtime workbench, a separate instance of Eclipse that can be configured and launched with any plug-ins under development to simulate the configuration of the target Eclipse installation. This configuration includes the Java Runtime Environment (JRE) to be used, as well as the set of plug-in projects in the workspace and installed plug-ins that should be available to the runtime workbench. The workbench could also be launched in debug mode to facilitate locating the cause of any defects in the plug-in.

In addition to the plug-ins comprising the IDE, the configuration used to test the system only included the plug-ins provided by the basic platform runtime as the JDT, PDE and any other plug-ins are not required in order to run the system. The Graphical Editing Framework (GEF) was later added to this configuration after it was chosen for use in the dependency graph view (section 6.3.3). After the first observation session, it was identified that the launching feature failed to run on the department machines because they used the 1.4 version of the JRE, rather than the newer version 5.0 (section 6.1.4). The JRE in the launch configuration was subsequently set to this version, in order that the system could be tested for correct functioning under this version prior to any further observation sessions.

Although this runtime workbench allowed the functionality of the system to be tested during the development and debugging of the system, it was not sufficient to use this test environment for testing prior to the release of a new increment. Firstly the system was required to work on multiple platforms, and Linux in particular. As the development was undertaken under Windows, testing using the runtime workbench

only considered this platform. Secondly, as the IDE plug-ins were only run from their respective projects in the workspace, this did not account for any packaging problems that might occur when building the individual plug-in jar files.

Therefore before each new release of the system, each plug-in was built and installed in a clean Eclipse platform installation. This was used as the environment for performing the tests. As it would not have been feasible to attempt to test the IDE on all of the platforms identified in the requirements analysis, the tests were only performed on installations under the Windows XP and SUSE Linux operating systems, given that the machines used for the demonstration and observations either ran under Windows or Linux.

7.2 Selected Examples of Tests

Let us now consider how the informal testing approach that was adopted was used to test certain aspects of the system.

7.2.1 Block Commenting

Given that the block commenting feature was a simple function, it was feasible to construct a small set of test cases within the timeframe of the dissertation to demonstrate that this function operated correctly under normal circumstances (Table 7.1). These tests were manually performed at the system level, for example by activating the menu item, rather than at the individual class level, as this is the level at which the function would need to work during the next evaluation session. The tests were restricted to black-box tests of normal usage, rather than trying to test the individual paths through the code, as this increased overhead would have slowed the development process. Given that the behaviour of the function was designed to work in the same way as the JDT (section 6.3.2), the test cases were devised from the behaviour of the JDT function in the same scenario.

Table 7.1: Block Commenting Test Cases

Test Case	Expected Result
Cursor on commented line	Comment character removed from start of line
Cursor on uncommented line	Comment character added to start of line
Multiple commented lines selected	Comment character removed from start of all lines. Text remains selected
Multiple uncommented lines selected	Comment character added to start of all lines. Text remains selected
Mixture of commented and uncommented lines selected	Comment character added to start of all lines. Text remains selected
Menu item selected	Comment is toggled
Keyboard shortcut entered	Comment is toggled

7.2.2 Data Structure

Unlike other system components such as syntax highlighting or launching LPARSE, the building of the data structure did not provide any visible feedback to indicate that it was functioning correctly. A `ContentOutlinePage` for the Eclipse outline view was therefore rapidly developed. Subclasses of this abstract class define how the document open in the current editor should be displayed in the view. The implementation used for testing the data structure wrapped the node tree with a model that could be displayed in a treeview component (Figure 7.1). Several source files were opened in the editor and their contents checked against the data structure representation to verify that this was constructed as expected.

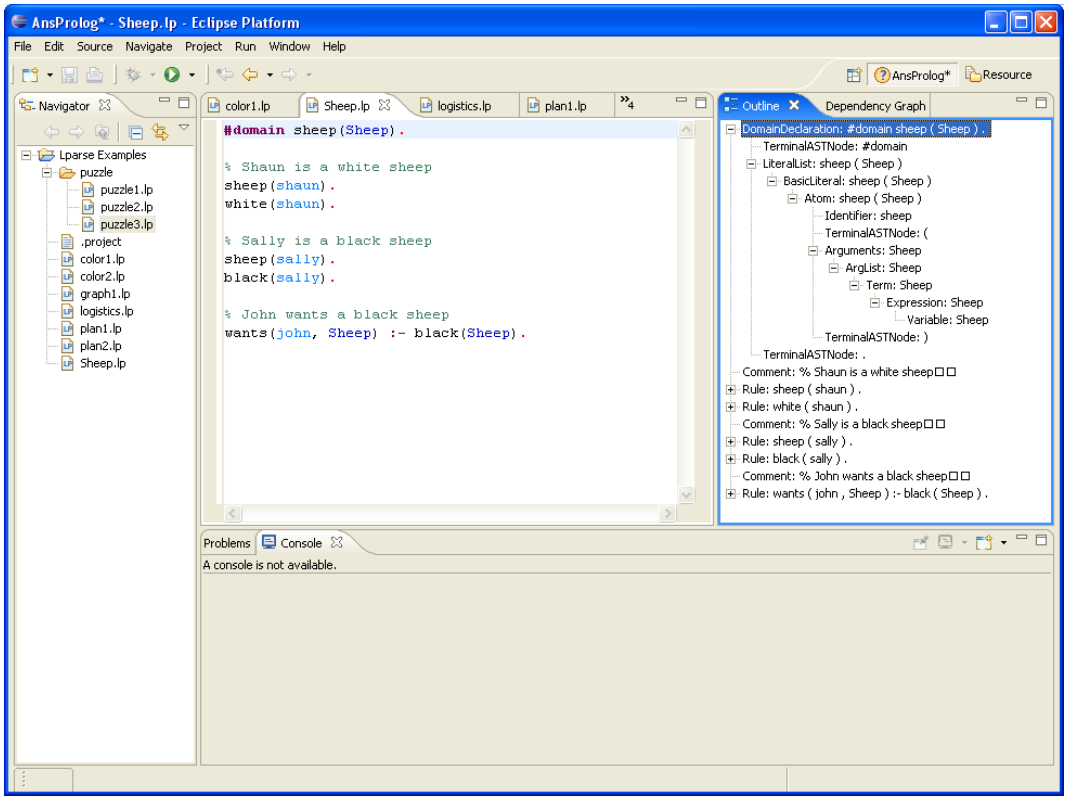


Figure 7.1: Outline View of Parsed Data Structure

7.3 Conclusion

Due to the time constraints of the dissertation it was not possible to rapidly develop and evaluate a wide range of features, and perform comprehensive unit and integration testing of the system. The functional testing performed was therefore limited to ensuring that the system could be used in the demonstrations and observations to further develop the requirements. This included limiting the testing of the system to the Windows and Linux operating systems.

However, it was identified that before the system could be delivered to the community as a usable tool, a more formal test approach would need to be undertaken to verify that the various features of the system do function correctly across all of the

target platforms, as well as when under stress or subjected to abnormal data inputs.

Chapter 8

Conclusion

The aim of this dissertation was to investigate how programming tools could be used to provide better support for Answer Set Programming, following the identification of the need for better programming tools by the Working group on Answer Set Programming. Let us consider what has been achieved in the course of this dissertation, and how this meets this aim, together with any further work that has been identified.

8.1 Identification of Required Tools

In order to provide better programming tools for ASP, it is necessary to identify what these tools are and understand the aspects of programming in this paradigm that require better support. In this dissertation a number of tools have been identified to support ASP, in the form of a set of requirements for an IDE, outlined in Appendix A.

Given the time and resource constraints of the dissertation, the elicitation and validation of these requirements was limited to discussions and demonstrations with users of ASP within the department at the University. Only a questionnaire was used to gather data from the wider community, and although this did provide a useful indication of the requirements of the IDE, it only received a response rate of 35%. With more resources available, a wider range of users would have been considered for the demonstrations and evaluations of the prototype system. This wider view would have allowed a better definition of the requirements of the IDE to be produced, by considering the needs of a more varied user base and any conflicts of interest between their different needs. Therefore this study should be extended to a wider user base, in order that the results obtained can be placed in a wider context.

One area of ASP requiring better support that was identified in the WASP report was that of debugging (Niemelä, 2005). The extremely high level of support given to debugging tools in the questionnaire that was distributed to the community (section 3.3.1), certainly suggests that debugging is indeed an aspect of ASP requiring better support, and potentially the aspect in which it is most needed. Although there is existing work in this area by Brain and De Vos (2005), Satoh (2000) and Syrjänen (n.d.b), further research into potential tools for supporting debugging should be performed together with their implementation in order that they can be

used in practice. Although such a study was beyond the scope of this dissertation, some potential tools were identified with the users such as the ability to display the grounding of a given rule and the rules used to generate an answer set.

In addition to identifying areas of ASP requiring better support and proposing tools to achieve this, the reverse approach was also adopted during this dissertation. That is to say tools that have been successfully used in other programming paradigms were investigated in order to consider how they could be adapted to ASP. Although the IDE itself was such a tool, one of the sub-tools considered in more detail was that of syntax highlighting. In particular, through the demonstration and evaluation of the prototype system a set of syntactic elements to be highlighted was proposed. However in order to validate whether this feature would benefit the programmer, and indeed identify which of the elements support or possibly even hinder the programming process, a study comparing the use of a highlighted and non-highlighted editor should be performed.

Another tool that was identified to be of potential use when programming in ASP was that of automatically indenting the code to facilitate maintaining a consistent, easy to read layout throughout the program. However, as discussed in section 5.3.1, the layout that the tool should adhere to would first need to be defined. Therefore it is proposed that a study into coding styles for ASP should be undertaken in order to define a common set of coding standards to improve the readability and maintainability of code.

8.2 Improvements to Existing Tools

In addition to the new ASP tools that were identified in the requirements elicitation process, an improvement to an existing tool was also identified. One requirement of the IDE that was raised throughout the elicitation process was for a tool to perform block commenting. This was due to the syntax of the LPARSE solver only supporting single line comments rendering the commenting of large blocks of code a tedious process. As discussed in section 6.1.4, although developing this tool would help support the programmer, it is resolving the problem in the wrong place. It would be better to add multi-line comments to the LPARSE syntax, in order that all ASP programmers could benefit from faster commenting, regardless of whether they use the IDE or not.

8.3 Improvements to Programming Practices

Adopting better programming practices are an alternative means to tools for facilitating the programming process. One observation that was made during the evaluation session was the use of single letter variable names, making it difficult to determine what the domain of the variable was and whether it had previously been used to represent something else (section 6.1.4). A suggested improvement to current practice was therefore to use more descriptive variable names, given that they are supported by the LPARSE syntax. However a wider investigation would need to be performed to determine whether the use of single letter variables is a common practice, and identify any other areas of ASP programming that could be improved.

8.4 Development of an IDE

In addition to identifying a set of requirements for the IDE, a working system implementing a subset of these was developed. Therefore the project has also helped to fill the gap in programming tools for ASP. Moreover the exploratory development approach that was adopted, used this tangible artefact to help elicit further requirements through a demonstration to the users and observations of system usage.

The AnsProlog* Programming Environment (APE) was implemented as a plug-in for the Eclipse platform as this met some of the requirements of the IDE (multi-platform support, plug-in mechanism) in addition to providing some basic functionality. This facilitated the rapid development of the system, enabling time to be spent exploring a wider range of features than if this basic functionality had to be implemented. The features implemented in the IDE are as follows:

- LPARSE Editor
  - Syntax highlighting
  - Error and warning underlining
  - Block commenting
- Launching LPARSE and SMODELS
  - Graphical user interface for configuring program arguments
  - Piping SMODELS output to another program
- Multiple file support
- Display of program dependency graph
- Multi-platform support (provided by Eclipse)
- Integrated version control tools (provided by Eclipse)
- Integrated build script support (provided by Eclipse)
- Support for integrating additional solvers into the IDE (provided by Eclipse)

Due to the size of the problem being investigated, it was anticipated at the start of the dissertation that it would not be possible to investigate and implement all of the requirements identified. Moreover, as we have just considered, some of the requirements require further detailed studies to be performed before they can be implemented. It was therefore expected that a full IDE would not be produced, however it is felt that as many features as possible were considered given the time constraints of the dissertation.

Further work would include implementing the other requirements identified, including the integration of the DLV solver into the system, as well as exploring further requirements. It is intended that the system will be made available to the wider community to aid the requirements elicitation process, and allow the users to make use of the features provided.

However as discussed in section 6.3.3, the dependency graph feature was not fully completed due to the limitations of the layout algorithm provided by the Draw2D library. Before this feature could be made available to the users, these limitations would need to be resolved by implementing a better layout algorithm or adopting a different graphics library.

Furthermore, the testing of the system that was performed was limited in order to allow a greater number of features to be explored. Therefore before the system could be released to the community as a fully functional tool, a more detailed testing approach would need to be adopted to ensure that the system would be reliable.

8.5 Summary

The dissertation has succeeded in its aim to investigate the area of programming tools for ASP. Not only has a set of requirements been defined, but a system implementing a subset of these features has been implemented. In addition, some improvements to existing tools and programming practices have been identified as alternatives to the provision of additional tools.

To further develop the system, the investigation of the requirements needs to be extended to the wider community, potentially through the release of the system to the users. However, more thorough system testing needs to be performed before this can be released as more than an evaluation tool. The need for further investigation into the areas of debugging, coding standards and programming practices for ASP has also been identified.

Bibliography

Anger, C., Konczak, K. and Linke, T. (2002). Nomore: Non-monotonic reasoning with logic programs, *JELIA '02: Proceedings of the European Conference on Logics in Artificial Intelligence*, Springer-Verlag, London, UK, pp. 521–524.

Appel, A. W. and Ginsburg, M. (2004). *Modern Compiler Implementation in C*, Press Syndicate of the University of Cambridge.

Apt, K. (2003). The logic programming paradigm and prolog, *in* J. Mitchell (ed.), *Concepts in Programming Languages*, Cambridge University Press, Cambridge, UK, pp. 475–508.

Arthorne, J. and Laffra, C. (2004). *Official Eclipse 3.0 FAQs*, Addison-Wesley, Boston, MA, USA.

Babovich, Y., Erdem, E. and Lifschitz, V. (2000). Fages' theorem and answer set programming, *in* C. Baral and M. Truszczynski (eds), *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning, NMR'2000*.

Baral, C. (2003). *Knowledge Representation, Reasoning and Declarative Problem Solving*, Cambridge University Press, Cambridge, UK.

Bevan, N. and Macleod, M. (1994). Usability measurement in context, *Behaviour and Information Technology* **13**: 132–145.

Bloch, J. (2001). *Effective Java Programming Language Guide*, Addison Wesley, Boston, MA, USA.

Boekhoudt, C. (2003). The big bang theory of ides, *Queue* **1**(7): 74–82.

Brain, M. and De Vos, M. (2004). Towards incremental computation for the answer set semantics: Preliminary report. Shortlisted poster at The Twenty-fourth SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence. Queen's College, Cambridge, UK (December 2004).

Brain, M. and De Vos, M. (2005). Debugging logic programs under the answer set semantics, *in* M. De Vos and A. Provetti (eds), *Answer Set Programming: Advances in Theory and Implementation*, Research Press International, pp. 142 – 152.

Bruckhaus, T., Madhavji, N. H., Janssen, I. and Henshaw, J. (1996). The impact of tools on software productivity, *IEEE Softw.* **13**(5): 29–38.

Calimeri, F. and Ianni, G. (2005). External sources of computation for answer set solvers, *Lecture Notes in Computer Science* **3662**: 105–118.

Clayberg, E. and Rubel, D. (2004). *Eclipse: Building Commercial-Quality Plug-ins*, Addison-Wesley, Boston, MA, USA.

Colmerauer, A. and Roussel, P. (1993). The birth of prolog, *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, ACM Press, New York, NY, USA, pp. 37–52.

Constantine, L. L. and Lockwood, L. A. D. (1999). *Software for use: a practical guide to the models and methods of usage-centered design*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Curley, C. (2002). Emacs: the free software ide, *Linux Journal* **2002**(98): 2.

Curtis, B. (1982). A review of human factors research on programming languages and specifications, *Proceedings of the 1982 conference on Human factors in computing systems*, ACM Press, New York, NY, USA, pp. 212–218.

Delisle, N. M., Menicosy, D. E. and Schwartz, M. D. (1984). Viewing a programming environment as a single tool, *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, ACM Press, New York, NY, USA, pp. 49–56.

Eclipse (2003). Eclipse platform technical overview.
URL: *http://www.eclipse.org/whitepapers/eclipse-overview.pdf*

Eclipse (2005a). Draft: Eclipse platform technical overview. Updated December 2005 for Eclipse 3.1.
URL: *http://www.eclipse.org/whitepapers/eclipse-overview.pdf*

Eclipse (2005b). Eclipse project plan 3.1 (final).
URL: *http://www.eclipse.org/eclipse/development/eclipse_project_plan_3_1.html #TargetOperatingEnvironments [Accessed 6 April 2006]*

Eclipse (2005c). Eclipse source builds (source from zip).
URL: *http://download.eclipse.org/eclipse/downloads/drops/R-3.1.2-20060118 1600/srcIncludedBuildInstructions.html [Accessed 7 April 2006]*

Eclipse (2005d). Platform plug-in developer guide.

Edgar, N., Haaland, K., Li, J. and Peter, K. (2004). Eclipse user interface guidelines.
URL: *http://www.eclipse.org/articles/Article-UI-Guidelines/Contents.html [Accessed 26 April 2006]*

Francez, N., Goldenberg, S., Pinter, R. Y., Tiomkin, M. and Tsur, S. (1985). An environment for logic programming, *Proceedings of the ACM SIGPLAN 85 symposium on Language issues in programming environments*, ACM Press, New York, NY, USA, pp. 179–190.

Free Software Foundation (1992). Gnu general public license version 2.
URL: *http://www.gnu.org/licenses/gpl.html [Accessed 22 April 2006]*

Fry, C. (1997). Programming on an already full brain, *Communications of the ACM* **40**(4): 55–64.

Geer, D. (2005). Eclipse becomes the dominant java ide, *Computer* **38**(7): 16–18.

Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming, *in* R. A. Kowalski and K. Bowen (eds), *Proceedings of the Fifth International Conference on Logic Programming*, The MIT Press, Cambridge, Massachusetts, pp. 1070–1080.

Genesereth, M. R. and Ginsberg, M. L. (1985). Logic programming, *Communications of the ACM* **28**(9): 933–941.

Glickstein, B. (1997). *Writing GNU Emacs Extensions*, O'Reilly & Associates Inc.

Heidt, M. L. (2001). *Developing an inference engine for aset-prolog*, Master's thesis, University of Texas at El Paso.

Ho, E. (2003). creating a text-based editor for eclipse, *Technical report*, Hewlett-Packard Company.
URL: *http://devresource.hp.com/drc/technical_white_papers/eclipeditor/Eclipse-Editor.pdf [Accessed 13 April 2006]*

Kolvekal, L. (2004). *Developing an inference engine for cr-prolog with preferences*, Master's thesis, Texas Tech University.

Komorowski, H. J. and Omori, S. (1985). A model and an implementation of a logic programming environment, *Proceedings of the ACM SIGPLAN 85 symposium on Language issues in programming environments*, ACM Press, New York, NY, USA, pp. 191–198.

Konczak, K., Schaub, T. and Linke, T. (2003). Graphs and colorings for answer set programming with preferences, *Fundamenta Informaticae* **57**(2-4): 393–421.

Lee, D. (2003). Display a uml diagram using draw2d, *Eclipse Corner Article* .
URL: *http://www.eclipse.org/articles/Article-GEF-Draw2d/GEF-Draw2d.html [Accessed 23 April 2006]*

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. (2004). The dlv system for knowledge representation and reasoning, *TOCL* .

Lierler, Y. and Maratea, M. (2004). Cmodels-2: Sat-based answer set solver enhanced to non-tight programs, *Lecture Notes in Computer Science* **2923**: 346–350.

Lin, F. and Zhao, Y. (2004). Assat: computing answer sets of a logic program by sat solvers, *Artificial Intelligence* **157**(1-2): 115–137.

Linke, T., Bösel, A., Anger, C., Gebser, M. and Konczak, K. (n.d.). nomore: A graph-based system for non-monotonic reasoning with logic programs under answer set semantics.
URL: *http://www.cs.uni-potsdam.de/ linke/nomore/ [Accessed 4 December 2005]*

LinuxQuestions.org (2006). Ide of the year.
URL: *http://www.linuxquestions.org/questions/showthread.php?t=409026 [Accessed 26 April 2006]*

Moore, A. and Redmond-Pyle, D. (1995). *Graphical User Interface Design and Evaluation: A Practical Process*, Prentice Hall PTR, Upper Saddle River, NJ, USA.

NetBeans (n.d.). Netbeans ide 5.0 plug-in module quick start guide.
URL: *http://platform.netbeans.org/tutorials/quickstart-nbm.html [Accessed 26 April 2006]*

Nielsen, J. and Molich, R. (1990). Heuristic evaluation of user interfaces, *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM Press, New York, NY, USA, pp. 249–256.

Niemelä, I. (ed.) (2005). *WASP WP3 Report: Language Extensions and Software Engineering for ASP.*
URL: *http://www.tcs.hut.fi/Research/Logic/wasp/wp3/wasp-wp3-web.pdf [Accessed 26 November 2005]*

Preece, J., Rogers, Y. and Sharp, H. (2002). *Interaction Design*, John Wiley & Sons, Inc., New York, NY, USA.

Reiss, S. P. (1996). Software tools and environments, *ACM Computing Surveys* **28**(1): 281–284.

Robinson, J. A. (1992). Logic and logic programming, *Communications of the ACM* **35**(3): 40–65.

Ruane, J. M. (2005). *Essentials of Research Methods: A Guide to Social Science Research*, Blackwell Publishing Ltd, Oxford, UK.

Satoh, K. (2000). Consistency management in software engineering by abduction, *Proceedings of the ICSE-2000 Workshop on Intelligent Software Engineering*, pp. 90 – 99.

Seeley, D. M. (2003). Coding smart: People vs. tools, *Queue* **1**(6): 33–40.

Sheil, B. A. (1981). The psychological study of programming, *ACM Comput. Surv.* **13**(1): 101–120.

Shneiderman, B. (1997). *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Sommerville, I. (2001). *Software Engineering*, 6th edn, Addison-Wesley Publishers Ltd., Harlow, England.

Stiller, E. and LeBlanc, C. (2002). *Project-Based Software Engineering: An Object-Oriented Approach*, Addison-Wesley, Boston, MA, USA.

Syrjänen, T. (1998). Implementation of local grounding for logic programs with stable model semantics, *Technical report*, Helsinki University of Technology.

Syrjänen, T. (n.d.a). *Lparse 1.0 User's Manual.*
URL: *http://www.tcs.hut.fi/Software/smodels/lparse.ps*

Syrjänen, T. (n.d.b). On debugging asp. draft version.

Syrjänen, T. and Niemelä (2001). The smodels system, *LPNMR '01: Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, Springer-Verlag, London, UK, pp. 434–438.

Szurszewski, J. (2003). We have lift-off: The launching framework in eclipse, *Eclipse Corner Article* .
URL: *http://www.eclipse.org/articles/Article-Launch-Framework/launch.html [Accessed 13 April 2006]*

Vaughan-Nichols, S. J. (2003). The battle over the universal java ide, *Computer* **36**(4): 21–23.

Weinberg, G. M. (1998). *The psychology of computer programming (silver anniversary ed.)*, Dorset House Publishing Co., Inc., New York, NY, USA.

Wilhelm, R. and Maurer, D. (1995). *Compiler Design*, Addison-Wesley Publishing Co. Inc., Wokingham, England.

Wolfe, A. (2003). Toolkit: Eclipse: A platform becomes an open-source woodstock, *Queue* **1**(8): 14–16.

Appendix A

Requirements Specification

The primary aim of this dissertation was to develop a set of requirements for an IDE for Answer Set Programming. Due to the exploratory development approach adopted, the requirements of the IDE were identified throughout the course of the dissertation and consequently are discussed throughout the dissertation.

For ease of reference they are collated here together with a reference to the sections in which they are discussed in more detail.

Non-Functional Requirements

1. The system should integrate existing ASP solvers into the environment
   - 1.1 The system should integrate the DLV and LPARSE/SMODELS solvers into the environment (3.2.1)
   - 1.2 The system should provide an extension mechanism to allow other solvers to be integrated into the environment (3.2.1)
2. The system should be able to run on multiple platforms
   - 2.1 The system must run on Linux (3.2.2)
   - 2.2 The system should run on the platforms supported by the underlying solvers (3.2.2)
3. The system should support ASP programs that are split over multiple files (3.3)

Functional Requirements

1. The system should provide a source file editor for the input language of any solvers integrated into the environment
   - 1.1 The editor should highlight the syntax of the source file (3.3)
     - 1.1.1 Keywords (3.3)
     - 1.1.2 Comments (5.2.1)

    1.1.3 Variables (5.2.1)
    1.1.4 Functions (5.2.1)
    1.1.5 Constants (5.2.1)
    1.1.6 Atoms (5.2.1)
    1.1.7 Negated atoms (5.3.1)
    1.1.8 ← operator (5.3.1)
    1.1.9 End of line operator (5.3.1)
  1.2 The editor should provide an autocompletion mechanism (3.3)
  1.3 The editor should provide automatic syntax checking
    1.3.1 The editor should underline syntax errors (3.3.2)
    1.3.2 The editor should underline warnings (5.3.1)
  1.4 The editor should automatically indent source code (5.3.1)
  1.5 The editor should provide bracket matching (5.3.1)
  1.6 The editor should provide block commenting (5.3.1)
2. The system should provide templates for new source files (6.1.4)
  2.1 The template should contain a header containing the following fields:
    2.1.1 Filename
    2.1.2 Description
    2.1.3 Author
    2.1.4 Email address
3. The system should be able to perform various analyses of the source programs
  3.1 The system should be able to represent a source program as a dependency graph (3.3)
  3.2 The system should be able to determine whether a program is tight or not (3.3.2)
4. The system should be able to convert programs between different syntaxes
  4.1 The system should be able to convert programs between the LPARSE and DLV syntaxes (3.3)
5. The system should support launching solvers from within the environment (3.3)
  5.1 The system should be able to filter the input to a solver (3.3.2)
  5.2 The system should be able to filter the output from a solver (3.3.2)
  5.3 The system should be able to save the output of a grounder (5.3.1)
  5.4 The system should be able to save any program output (5.3.1)
  5.5 The system should provide a graphical user interface for configuring solver arguments (5.3.1)
  5.6 The system should wrap any textual solver output to fit on the screen (6.2.4)
  5.7 The system should be able to calculate benchmarks for each solver

5.7.1 Computation time (3.3.2)

6. The system should provide debugging tools (3.3)

   6.1 The system should be able to compute the grounding of a given rule (3.3)

   6.2 The system should provide a tracing facility for computing models bottom-up (3.3.2)

   6.3 The system should be able to display the rules used to generate an answer set (3.3.2)

7. The system should provide version control tools (3.3)

8. The system should provide integrated support for build scripts (3.3.2)

Appendix B

Questionnaire

In order to elicit the key requirements of the system and some potential features, a questionnaire was developed and sent out to members of the ASP community. This process together with an analysis of the obtained results is discussed in more detail in Chapter 3.

In this appendix we present a copy of the questionnaire, together with a summary of the results obtained.

Questionnaire: An IDE for ASP

1. Which ASP tools do you currently use?

   ☐ lparse/smodels ☐ dlt ☐ Other(s) ……

   ☐ dlv ☐ noMoRe

2. Which operating systems do you use for ASP development?

   ☐ Unix ☐ Windows ☐ Other(s) ……

   ☐ Linux ☐ Solaris

   ☐ MacOS

3. How many years experience of ASP development do you have? ……

4. If an IDE for ASP were to be developed, would you be interested in using it? Yes/No

5. Which of the following features would you like to see in an IDE for ASP?

   Please indicate the importance of any features you are interested in:
   0 (least important) – 10 (most important])

   ☐ Integration of editor and lparse/smodels

   ☐ Integration of editor and dlv

   ☐ Syntax/predicate highlighting

   ☐ Automatic completion of predicates

   ☐ Ability to associate a textual description with each predicate

   ☐ Debugging tools

   ☐ Integrated version control tools

   ☐ Modularity of programs over multiple files

   ☐ Automatic file conversion between lparse and dlv

   ☐ Graph representation of programs

   ☐ Replacement of a rule by its grounding

   ☐ Other(s) ……

   ……

   ……

6. Do you have any other comments regarding an IDE for ASP?

   ……

   ……

   ……

7. Are you familiar with the Eclipse development platform[1]? Yes/No

8. Would you have any objections to an IDE for ASP built on this framework, or prefer a different framework?

   ……

   ……

   ……

[1] www.eclipse.org

Results Summary

1. Which ASP tools do you currently use?

Tool	No. of Responses
lparse/smodels	11
dlv	15
dlt	3
noMoRe	2
lparse/cmodels	2
dlv-ex	1
CR-Prolog	2
ASSAT	1
ASET-Prolog	1

2. Which operating systems do you use for ASP development?

Operating System	No. of Responses
Unix	2
Linux	15
MacOS	4
Windows	5
Solaris	4

3. How many years experience of ASP development do you have?

Years	No. of Responses
1	1
2	3
3	5
4	2
5	2
7	2
10	1
No response	1

4. If an IDE for ASP were to be developed, would you be interested in using it?

Response	No. of Responses
Yes	16
No	0
No Response	1

5. Which of the following features would you like to see in an IDE for ASP?

Feature	Yes	No	0	1	2	3	4	5	6	7	8	9	10
Integration of editor and lparse /smodels	1	2	1	0	2	0	0	2	3	1	4	1	0
Integration of editor and dlv	2	1	1	0	2	0	0	2	1	1	2	0	5
Syntax/predicate highlighting	3	0	1	0	1	1	2	1	2	1	2	3	0
Automatic completion of predicates	2	1	0	2	1	1	2	1	2	2	2	1	0
Ability to associate a textual description with each predicate	2	1	1	0	1	1	1	3	1	2	2	2	0
Debugging tools	3	0	0	0	0	0	0	1	1	0	1	4	7
Integrated version control tools	2	1	2	2	0	0	0	4	0	4	1	0	1
Modularity of programs over multiple files	2	1	1	0	0	0	1	1	2	3	4	0	2
Automatic file conversion between lparse and dlv	2	1	0	0	0	1	0	0	0	1	4	4	4
Graph representation of programs	3	0	0	0	1	0	0	3	1	2	5	2	0
Replacement of a rule by its grounding	0	3	0	0	0	0	2	2	4	1	2	2	1

Other Features

- Statical analysis of program (non-)tightness, e.g.: strongly connected components of positive dependency graph, properties of connecting rules (like number of scc atoms in rules' bodies)
- Some support for make files and input / output filter scripts.
- Integrated tools for Program (i.e. rule) rewriting in order to automate translation among different syntaxes
- Automatic syntax check
- The possibility to choose which solver one wants to use (eg, with a checkbox, or dropdown menu thing)
- Also nice would be a feature that allows to do some benchmarks, eg, "for this program, what is the timing for dlv, what is the timing for smodels"
- If this system is meant to be a full IDE it would be nice to have a plugin system that will enable it to be extended to other aprolog inference systems.
- 'Tracing" facility for computing models bottom up
- Display of generating rules
- Display of components in dependency graph
- Display of dependencies

6. Do you have any other comments regarding an IDE for ASP?

On the last point above, you can see that it seems important to me that an IDE for ASP provides structural information about programs (like tightness). This involves not only atoms or predicates but also rules' bodies (e.g.: unique or occuring in

multiple rules). From recent experience, I can say that this is important for figuring out significant program properties. I conjecture that disjunctive heads (e.g. head cycles) and aggregates play a role as well.

I could be useful for Rapid Application Development of ASP programs

7. Are you familiar with the Eclipse development platform?

Response	No. of Responses
Yes	5
No	7
No Response	5

8. Would you have any objections to an IDE for ASP built on this framework, or prefer a different framework?

No

I'd prefer emacs :-D seriously, Eclipse seems to be the way of doing things these days. Does building / running Eclipse require non free Java tools?

No objections

I think that an IDE built over Eclipse will be very interesting, since it is a widely known framework and a lot of plug-ins already exist that may be exploited

No objections

No. I used Eclipse a long time ago and it seemed ok.

No objections

No real preference so long as the system works well and is simple to use.

No objections

Appendix C

Source Code

The full source code for each of the individual AnsProlog* Programming Environment plug-ins is included on the CD accompanying this dissertation.

However for ease of reference, the source code of some of the key classes described in the dissertation is included in this Appendix:

- `uk.ac.bath.cs.asp.ide.lparse.internal.ui.editor.LparseRuleScanner`
- `uk.ac.bath.cs.asp.ide.ui.editor.syntax.SyntaxColoringElement`
- `uk.ac.bath.cs.asp.ide.internal.ui.preferences.SyntaxColoringPreferences`
- `uk.ac.bath.cs.asp.ide.lparse.core.ast.LparseSourceProblem`
- `uk.ac.bath.cs.asp.ide.lparse.internal.ui.editor.LparseReconcilingStrategy`
- `uk.ac.bath.cs.asp.ide.smodels.launch.SmodelsLaunchConfigurationDelegate`
- `uk.ac.bath.cs.asp.ide.lparse.core.ast.LparseSourceFile`
- `uk.ac.bath.cs.asp.ide.lparse.internal.ui.editor.actions.ToggleCommentAction`
- `uk.ac.bath.cs.asp.ide.lparse.internal.dependencygraph.DependencyGraphBuilder`
- `uk.ac.bath.cs.asp.ide.dependencygraphs.ui.views.DependencyGraphView`

```
/*****************************************************************************
 * APE: AnsProlog* Programming Environment plug-in for the Eclipse platform
 * Copyright (C) 2006  Adrian Sureshkumar
 *
 * This program is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License as published by the Free Software
 * Foundation; either version 2 of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT ANY
 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE.  See the GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along with this
 * program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street,
 * Fifth Floor, Boston, MA  02110-1301, USA.
 *****************************************************************************/

package uk.ac.bath.cs.asp.ide.lparse.internal.ui.editor;

import java.util.LinkedList;

import org.eclipse.jface.text.TextAttribute;
import org.eclipse.jface.text.rules.EndOfLineRule;
import org.eclipse.jface.text.rules.ICharacterScanner;
import org.eclipse.jface.text.rules.IRule;
import org.eclipse.jface.text.rules.IToken;
import org.eclipse.jface.text.rules.IWordDetector;
import org.eclipse.jface.text.rules.RuleBasedScanner;
import org.eclipse.jface.text.rules.Token;
import org.eclipse.jface.text.rules.WordRule;
import org.eclipse.jface.text.source.ISourceViewer;
import org.eclipse.swt.SWT;
import org.eclipse.swt.graphics.Color;
import org.eclipse.swt.graphics.RGB;

import uk.ac.bath.cs.asp.ide.lparse.core.ast.LparseSourceFile;
import uk.ac.bath.cs.asp.ide.ui.editor.syntax.ISyntaxColoringConfiguration;
import uk.ac.bath.cs.asp.ide.ui.editor.syntax.SyntaxColoringConfigurationListener;
import uk.ac.bath.cs.asp.ide.ui.editor.syntax.SyntaxColoringElement;

/**
 * A rule based scanner for syntax highlighting of lparse source files
 * @author Adrian Sureshkumar
 */
public class LparseRuleScanner extends RuleBasedScanner
{
    private final ColorManager colorManager;
    private final ISourceViewer sourceViewer;
    private final ISyntaxColoringConfiguration coloringConfig;

    //Keywords
    private static final String NOT_KEYWORD = "not";
    private static final String[] STATEMENT_KEYWORDS = new String[] {"compute",
        "maximize", "minimize"};
    private static final String[] DECLARATION_KEYWORDS = new String[] {"const",
        "domain", "external", "function", "hide", "option", "show", "weight"};

    //Identifiers
    private String[] atoms = new String[0];
    private String[] functions = new String[0];
    private String[] symbolicFunctions = new String[0];
    private String[] numericConstants = new String[0];
    private String[] symbolicConstants = new String[0];

    //Tokens
    final Token defaultText;
    final Token comment;
    final Token keyword;
    final Token declarationKeyword;
    final Token statementKeyword;
    final Token variable;
    final Token implication;
    final Token dot;
    final Token number;
    final Token atom;
    final Token numericConstant;
    final Token symbolicConstant;
    final Token function;
    final Token symbolicFunction;

    /**
     * Creates a new scanner
     * @param colorManager The color manager
     * @param coloringConfig The syntax coloring configuration
     * @param sourceViewer The source viewer
     */
    public LparseRuleScanner(final ColorManager colorManager,
            final ISyntaxColoringConfiguration coloringConfig,
            final ISourceViewer sourceViewer)
    {
        this.colorManager = colorManager;
        this.sourceViewer = sourceViewer;
        this.coloringConfig = coloringConfig;

        //Create token types
        defaultText = new SyntaxColoringToken(SyntaxColoringElement.DEFAULT_TEXT);
        comment = new SyntaxColoringToken(SyntaxColoringElement.COMMENT);
        keyword = new SyntaxColoringToken(SyntaxColoringElement.KEYWORD);
        declarationKeyword = new
SyntaxColoringToken(SyntaxColoringElement.DECLARATION_KEYWORD);
        statementKeyword = new
SyntaxColoringToken(SyntaxColoringElement.STATEMENT_KEYWORD);
        variable = new SyntaxColoringToken(SyntaxColoringElement.VARIABLE);
        implication = new
SyntaxColoringToken(SyntaxColoringElement.IMPLICATION_OPERATOR);
        dot = new SyntaxColoringToken(SyntaxColoringElement.END_OF_LINE_DOT);
        number = new SyntaxColoringToken(SyntaxColoringElement.NUMBER);
        atom = new SyntaxColoringToken(SyntaxColoringElement.ATOM);
        numericConstant = new
SyntaxColoringToken(SyntaxColoringElement.NUMERIC_CONSTANT);
        symbolicConstant = new
SyntaxColoringToken(SyntaxColoringElement.SYMBOLIC_CONSTANT);
        function = new SyntaxColoringToken(SyntaxColoringElement.FUNCTION);
        symbolicFunction = new
SyntaxColoringToken(SyntaxColoringElement.SYMBOLIC_FUNCTION);
        setDefaultReturnToken(defaultText);

        createRules();
    }

    /**
     * Create the syntax coloring rules
     */
    private void createRules()
    {
        final LinkedList rules = new LinkedList();

        final IWordDetector wordDetector = new IWordDetector()
        {
            public boolean isWordStart(char c)
            {
                return isAToZ(c) || isaToz(c);
            }

            public boolean isWordPart(char c)
            {
```

```
            return  isAToZ(c) || isaToz(c) || is0To9(c) || c == '_';
        }
    };

    //------------------------  Comments  ------------------------\\

    rules.add(new EndOfLineRule("%", comment));

    //------------------------  Keywords  ------------------------\\

    //Keywords used in declaration
    final WordRule declarationKeywordRule = new WordRule(new IWordDetector()
    {
        public boolean isWordStart(char c)
        {
            return (c == '#' || isaToz(c));
        }

        public boolean isWordPart(char c)
        {
            return isaToz(c);
        }
    });
    for (int i = 0; i < DECLARATION_KEYWORDS.length; i++)
    {
        declarationKeywordRule.addWord(DECLARATION_KEYWORDS[i],
declarationKeyword);
        declarationKeywordRule.addWord('#' + DECLARATION_KEYWORDS[i],
declarationKeyword);
    }
    rules.add(declarationKeywordRule);

    //Keywords used in statements
    final WordRule statementKeywordRule = new WordRule(wordDetector);
    for (int i = 0; i < STATEMENT_KEYWORDS.length; i++)
    {
        statementKeywordRule.addWord(STATEMENT_KEYWORDS[i], statementKeyword);
    }
    rules.add(statementKeywordRule);

    //Other keywords
    final WordRule keywordRule = new WordRule(wordDetector);
    keywordRule.addWord(NOT_KEYWORD, keyword);
    rules.add(keywordRule);

    //------------------------  Variables  ------------------------\\

    final IWordDetector variableDetector = new IWordDetector()
    {
        public boolean isWordStart(char c)
        {
            return isAToZ(c);
        }

        public boolean isWordPart(char c)
        {
            return  isAToZ(c) || isaToz(c) || is0To9(c) || c == '_';
        }
    };
    final WordRule variableRule = new WordRule(variableDetector, variable);
    rules.add(variableRule);

    //------------------------  Identifiers  ------------------------\\

    final IWordDetector identifierDetector = new IWordDetector()
    {
        public boolean isWordStart(char c)
        {
```



```
            return isaToz(c) || c == '_';
        }

        public boolean isWordPart(char c)
        {
            return  isAToZ(c) || isaToz(c) || is0To9(c) || c == '_' || c == '\'';
        }
    };

    final WordRule identifierRule = new WordRule(identifierDetector, defaultText);
    for (int i = 0; i < atoms.length; i++)
    {
        identifierRule.addWord(atoms[i], atom);
    }
    for (int i = 0; i < numericConstants.length; i++)
    {
        identifierRule.addWord(numericConstants[i], numericConstant);
    }
    for (int i = 0; i < symbolicConstants.length; i++)
    {
        identifierRule.addWord(symbolicConstants[i], symbolicConstant);
    }
    for (int i = 0; i < functions.length; i++)
    {
        identifierRule.addWord(functions[i], function);
    }
    for (int i = 0; i < symbolicFunctions.length; i++)
    {
        identifierRule.addWord(symbolicFunctions[i], symbolicFunction);
    }
    rules.add(identifierRule);

    //------------------------   Numbers   ------------------------\\

    final IWordDetector numberDetector = new IWordDetector()
    {
        public boolean isWordStart(char c)
        {
            return is0To9(c);
        }

        public boolean isWordPart(char c)
        {
            return  is0To9(c);
        }
    };
    final WordRule numberRule = new WordRule(numberDetector, number);
    rules.add(numberRule);

    //------------------------  Operators  ------------------------\\
    final IRule implicationRule = new IRule()
    {
        public IToken evaluate(final ICharacterScanner scanner)
        {
            char c = (char)scanner.read();
            if(c == ':')
            {
                c = (char)scanner.read();
                if(c == '-')
                {
                    return implication;
                }
                else
                {
                    scanner.unread();
                    scanner.unread();
                    return Token.UNDEFINED;
                }
```

```
            }
            else
            {
                scanner.unread();
                return Token.UNDEFINED;
            }
        }
    };
    rules.add(implicationRule);

    final IWordDetector dotDetector = new IWordDetector()
    {
        public boolean isWordStart(char c)
        {
            return c == '.';
        }

        public boolean isWordPart(char c)
        {
            return c == '.';
        }
    };
    final WordRule dotRule = new WordRule(dotDetector, dot);
    dotRule.addWord("..", defaultText);
    rules.add(dotRule);

    //--------- Add the rules  ---------
    setRules((IRule[])rules.toArray(new IRule[rules.size()]));
}

/**
 * Callback method called when the model of the source file has been updated
 * @param model The updated model of the source file
 */
void modelUpdated(final LparseSourceFile model)
{
    atoms = model.getAtomTable().getSymbols();
    numericConstants = model.getNumericConstantTable().getSymbols();
    symbolicConstants = model.getSymbolicConstantTable().getSymbols();
    functions = model.getFunctionTable().getSymbols();
    symbolicFunctions = model.getSymbolicFunctionTable().getSymbols();

    //Create the new rules
    createRules();

    //Invalidate the current colouring
    sourceViewer.invalidateTextPresentation();
}

private boolean isaToz(char c)
{
    return (c >= 'a' && c <= 'z');
}

private boolean isAToZ(char c)
{
    return (c >= 'A' && c <= 'Z');
}

private boolean is0To9(char c)
{
    return (c >= '0' && c <= '9');
}

/**
 * A Token for syntax coloring which updates itself when the syntax coloring
 * configuration changes
 */
```

```
  private class SyntaxColoringToken extends Token
  {
      private final SyntaxColoringElement element;

      private Color color;
      private boolean isBold;
      private boolean isItalic;
      private boolean isStrikethrough;
      private boolean isUnderline;

      public SyntaxColoringToken(final SyntaxColoringElement element)
      {
          super(new Object());
          this.element = element;

          //Get the color
          color = colorManager.getColor(element, coloringConfig.getColor(element));

          //Get the text style
          isBold = coloringConfig.isBold(element);
          isItalic = coloringConfig.isItalic(element);
          isStrikethrough = coloringConfig.isStrikethrough(element);
          isUnderline = coloringConfig.isUnderline(element);

          updateTokenData();

          //Listen for changes in the syntax coloring
          coloringConfig.addSyntaxColoringConfigurationListener(new
SyntaxColoringConfigurationListener()
          {
              public void colorUpdated(SyntaxColoringElement element, RGB color)
              {
                  if(SyntaxColoringToken.this.element.equals(element))
                  {
                      SyntaxColoringToken.this.color = colorManager.getColor(element,
color);
                      updateTokenData();
                      sourceViewer.invalidateTextPresentation();
                  }
              }

              public void boldUpdated(SyntaxColoringElement element, boolean bold)
              {
                  if(SyntaxColoringToken.this.element.equals(element))
                  {
                      SyntaxColoringToken.this.isBold = bold;
                      updateTokenData();
                      sourceViewer.invalidateTextPresentation();
                  }
              }

              public void italicUpdated(SyntaxColoringElement element, boolean
italic)
              {
                  if(SyntaxColoringToken.this.element.equals(element))
                  {
                      SyntaxColoringToken.this.isItalic = italic;
                      updateTokenData();
                      sourceViewer.invalidateTextPresentation();
                  }
              }

              public void strikethroughUpdated(SyntaxColoringElement element, boolean
strikethrough)
              {
                  if(SyntaxColoringToken.this.element.equals(element))
                  {
                      SyntaxColoringToken.this.isStrikethrough = strikethrough;
```

```
                    updateTokenData();
                    sourceViewer.invalidateTextPresentation();
                }
            }

            public void underlineUpdated(SyntaxColoringElement element, boolean
underline)
            {
                if(SyntaxColoringToken.this.element.equals(element))
                {
                    SyntaxColoringToken.this.isUnderline = underline;
                    updateTokenData();
                    sourceViewer.invalidateTextPresentation();
                }
            }
        });
    }

    /**
     * Gets the text style of the token
     * @return The text style
     */
    private int getTextStyle()
    {
        int style = SWT.NONE;
        if(isBold) style |= SWT.BOLD;
        if(isItalic) style |= SWT.ITALIC;
        if(isStrikethrough) style |= TextAttribute.STRIKETHROUGH;
        if(isUnderline) style |= TextAttribute.UNDERLINE;

        return style;
    }

    private void updateTokenData()
    {
        setData(new TextAttribute(color, null, getTextStyle()));
    }
}
}
```



```
/*****************************************************************************
 * APE: AnsProlog* Programming Environment plug-in for the Eclipse platform
 * Copyright (C) 2006  Adrian Sureshkumar
 *
 * This program is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License as published by the Free Software
 * Foundation; either version 2 of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT ANY
 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE.  See the GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along with this
 * program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street,
 * Fifth Floor, Boston, MA  02110-1301, USA.
 ****************************************************************************/

package uk.ac.bath.cs.asp.ide.ui.editor.syntax;

import java.lang.reflect.Field;
import java.lang.reflect.Modifier;
import java.util.LinkedList;
import java.util.List;

import uk.ac.bath.cs.asp.ide.ui.ASPUIPlugin;

/**
 * A type-safe enum of syntax coloring elements
 * @author Adrian Sureshkumar
 */
public class SyntaxColoringElement
{
    public static final SyntaxColoringElement ATOM = new SyntaxColoringElement("Atom");
    public static final SyntaxColoringElement NUMERIC_CONSTANT = new
SyntaxColoringElement("Numeric constant");
    public static final SyntaxColoringElement SYMBOLIC_CONSTANT = new
SyntaxColoringElement("Symbolic constant");
    public static final SyntaxColoringElement FUNCTION = new
SyntaxColoringElement("Function");
    public static final SyntaxColoringElement SYMBOLIC_FUNCTION = new
SyntaxColoringElement("Symbolic Function");
    public static final SyntaxColoringElement COMMENT = new
SyntaxColoringElement("Comment");
    public static final SyntaxColoringElement KEYWORD = new
SyntaxColoringElement("Keyword");
    public static final SyntaxColoringElement DECLARATION_KEYWORD = new
SyntaxColoringElement("Keyword used in a declaration");
    public static final SyntaxColoringElement STATEMENT_KEYWORD = new
SyntaxColoringElement("Keyword used in a statement");
    public static final SyntaxColoringElement IMPLICATION_OPERATOR = new
SyntaxColoringElement("If operator");
    public static final SyntaxColoringElement END_OF_LINE_DOT = new
SyntaxColoringElement("End of statement dot");
    public static final SyntaxColoringElement VARIABLE = new
SyntaxColoringElement("Variable");
    public static final SyntaxColoringElement DEFAULT_TEXT = new
SyntaxColoringElement("Default Text");
    public static final SyntaxColoringElement NUMBER = new
SyntaxColoringElement("Number");

    private static final List elementList = new LinkedList();

    static
    {
        //Initialise the list of element types
        final Field[] fields = SyntaxColoringElement.class.getFields();

        for (int i = 0; i < fields.length; i++)
```

```
        {
            final Field field = fields[i];
            final int modifiers = field.getModifiers();

            //Add to list if its is a public element type constant
            if(field.getType().equals(SyntaxColoringElement.class) &&
                    Modifier.isPublic(modifiers) && Modifier.isStatic(modifiers)
                    && Modifier.isFinal(modifiers))
            {
                try
                {
                    elementList.add(field.get(null));
                }
                catch (Exception e)
                {
                    ASPUIPlugin.getLogger().logException(e);
                }
            }
        }
    }

    /**
     * Gets an array of syntax coloring elements
     * @return the syntax coloring elements
     */
    public static SyntaxColoringElement[] getElementList()
    {
        return (SyntaxColoringElement[])elementList.toArray(
                new SyntaxColoringElement[elementList.size()]);
    }

    private String name;

    private SyntaxColoringElement(final String name)
    {
        this.name = name;
    }

    public String toString()
    {
        return name;
    }
}
```



```
/*****************************************************************************
 * APE: AnsProlog* Programming Environment plug-in for the Eclipse platform
 * Copyright (C) 2006  Adrian Sureshkumar
 *
 * This program is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License as published by the Free Software
 * Foundation; either version 2 of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT ANY
 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE.  See the GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along with this
 * program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street,
 * Fifth Floor, Boston, MA  02110-1301, USA.
 *****************************************************************************/

package uk.ac.bath.cs.asp.ide.internal.ui.preferences;

import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

import org.eclipse.jface.preference.IPreferenceStore;
import org.eclipse.jface.preference.PreferenceConverter;
import org.eclipse.jface.util.IPropertyChangeListener;
import org.eclipse.jface.util.PropertyChangeEvent;
import org.eclipse.swt.graphics.RGB;

import uk.ac.bath.cs.asp.ide.ui.ASPUIPlugin;
import uk.ac.bath.cs.asp.ide.ui.editor.syntax.SyntaxColoringElement;

/**
 * Class to access syntax coloring preferences from the preference store
 * @author Adrian Sureshkumar
 */
public final class SyntaxColoringPreferences
{
    //Hide default constructor
    private SyntaxColoringPreferences() {}

    private static final IPreferenceStore PREFERENCE_STORE =
        ASPUIPlugin.getDefault().getPreferenceStore();

    private static final String COLOR_ATTRIBUTE = "color";
    private static final String BOLD_ATTRIBUTE = "bold";
    private static final String ITALIC_ATTRIBUTE = "italic";
    private static final String STRIKETHROUGH_ATTRIBUTE = "strikethrough";
    private static final String UNDERLINE_ATTRIBUTE = "underline";

    private static final String SYNTAX_PREFERENCE_ROOT =
        ASPUIPlugin.class.getPackage().getName() + ".editor.syntax";

    private static List listeners = new LinkedList();

    static
    {
        //Listen to changes in the preferences
        PREFERENCE_STORE.addPropertyChangeListener(new IPropertyChangeListener()
        {
            public void propertyChange(PropertyChangeEvent event)
            {
                //Get the preference that changed
                final String preference = event.getProperty();

                //Verify that it's a syntax coloring preference
                if(preference.startsWith(SYNTAX_PREFERENCE_ROOT) &&
                        !preference.equals(SYNTAX_PREFERENCE_ROOT))
```

```
                {
                    //Break up into element and attribute
                    final String[] subPreferences =
                        preference.substring(SYNTAX_PREFERENCE_ROOT.length() +
1).split("\\.");

                    if(subPreferences.length >= 2)
                    {
                        //Get the corresponding element
                        final String elementType = subPreferences[0];
                        SyntaxColoringElement element = null;

                        final SyntaxColoringElement[] elements =
SyntaxColoringElement.getElementList();
                        for (int i = 0; i < elements.length && element == null; i++)
                        {
                            if(elements[i].toString().toLowerCase().equals(elementType)
)
                                element = elements[i];
                        }

                        if(element != null)
                        {
                            //Notify listeners
                            final String attribute = subPreferences[1];

                            if(attribute.equals(COLOR_ATTRIBUTE))
                            {
                                fireColorUpdated(element, (RGB)(event.getNewValue()));
                            }
                            else if(attribute.equals(BOLD_ATTRIBUTE))
                            {
                                fireBoldUpdated(element,
((Boolean)event.getNewValue()).booleanValue());
                            }
                            else if(attribute.equals(ITALIC_ATTRIBUTE))
                            {
                                fireItalicUpdated(element,
((Boolean)event.getNewValue()).booleanValue());
                            }
                            else if(attribute.equals(STRIKETHROUGH_ATTRIBUTE))
                            {
                                fireStrikethroughUpdated(element,
((Boolean)event.getNewValue()).booleanValue());
                            }
                            else if(attribute.equals(UNDERLINE_ATTRIBUTE))
                            {
                                fireUnderlineUpdated(element,
((Boolean)event.getNewValue()).booleanValue());
                            }
                        }
                    }
                }
            }
        });
    }

    /**
     * Gets the color of the specified syntax element from the preference store
     * @param element The syntax element
     * @return The color of the syntax element
     */
    public static RGB getColor(final SyntaxColoringElement element)
    {
        return PreferenceConverter.getColor(PREFERENCE_STORE,
getColorPreferenceKey(element));
    }

    /**
     * Gets whether the specified syntax element is emboldened from the preference
store
     * @param element The syntax element
     * @return Whether the syntax element is emboldened
     */
    public static boolean getBold(final SyntaxColoringElement element)
    {
        return PREFERENCE_STORE.getBoolean(getBoldPreferenceKey(element));
    }

    /**
     * Gets whether the specified syntax element is italicized from the preference
store
     * @param element The syntax element
     * @return Whether the syntax element is italicized
     */
    public static boolean getItalic(final SyntaxColoringElement element)
    {
        return PREFERENCE_STORE.getBoolean(getItalicPreferenceKey(element));
    }

    /**
     * Gets whether the specified syntax element is struckthrough from the preference
store
     * @param element The syntax element
     * @return Whether the syntax element is struckthrough
     */
    public static boolean getStrikethrough(final SyntaxColoringElement element)
    {
        return PREFERENCE_STORE.getBoolean(getStrikethroughPreferenceKey(element));
    }

    /**
     * Gets whether the specified syntax element is underlined from the preference
store
     * @param element The syntax element
     * @return Whether the syntax element is underlined
     */
    public static boolean getUnderline(final SyntaxColoringElement element)
    {
        return PREFERENCE_STORE.getBoolean(getUnderlinePreferenceKey(element));
    }

    /**
     * Gets the default color of the specified syntax element from the preference store
     * @param element The syntax element
     * @return The default color of the syntax element
     */
    public static RGB getDefaultColor(final SyntaxColoringElement element)
    {
        return PreferenceConverter.getDefaultColor(PREFERENCE_STORE,
getColorPreferenceKey(element));
    }

    /**
     * Gets whether the specified syntax element is emboldened by default from the
preference store
     * @param element The syntax element
     * @return Whether the syntax element is emboldened by default
     */
    public static boolean getDefaultBold(final SyntaxColoringElement element)
    {
        return PREFERENCE_STORE.getDefaultBoolean(getBoldPreferenceKey(element));
    }

    /**
     * Gets whether the specified syntax element is italicized by default from the
```

```
preference store
     * @param element The syntax element
     * @return Whether the syntax element is italicized by default
     */
    public static boolean getDefaultItalic(final SyntaxColoringElement element)
    {
        return PREFERENCE_STORE.getDefaultBoolean(getItalicPreferenceKey(element));
    }

    /**
     * Gets whether the specified syntax element is struckthrough by default  from the
preference store
     * @param element The syntax element
     * @return Whether the syntax element is struckthrough by default
     */
    public static boolean getDefaultStrikethrough(final SyntaxColoringElement element)
    {
        return
PREFERENCE_STORE.getDefaultBoolean(getStrikethroughPreferenceKey(element));
    }

    /**
     * Gets whether the specified syntax element is underlined by default from the
preference store
     * @param element The syntax element
     * @return Whether the syntax element is underlined by defaults
     */
    public static boolean getDefaultUnderline(final SyntaxColoringElement element)
    {
        return PREFERENCE_STORE.getDefaultBoolean(getUnderlinePreferenceKey(element));
    }

    /**
     * Sets the color of the specified syntax element in the preference store
     * @param element The syntax element
     * @param color The color of the syntax element
     */
    public static void setColor(final SyntaxColoringElement element, final RGB color)
    {
        PreferenceConverter.setValue(PREFERENCE_STORE, getColorPreferenceKey(element),
color);
    }

    /**
     * Sets whether the specified syntax element is emboldened in the preference store
     * @param element The syntax element
     * @param bold Whether the syntax element is emboldened
     */
    public static void setBold(final SyntaxColoringElement element, final boolean bold)
    {
        PREFERENCE_STORE.setValue(getBoldPreferenceKey(element), bold);
    }

    /**
     * Sets whether the specified syntax element is italicized in the preference store
     * @param element The syntax element
     * @param italic Whether the syntax element is italicized
     */
    public static void setItalic(final SyntaxColoringElement element, final boolean
italic)
    {
        PREFERENCE_STORE.setValue(getItalicPreferenceKey(element), italic);
    }

    /**
     * Sets whether the specified syntax element is struckthrough in the preference
store
     * @param element The syntax element
     * @param strikethrough Whether the syntax element is struckthrough
     */
    public static void setStrikethrough(final SyntaxColoringElement element, final
boolean strikethrough)
    {
        PREFERENCE_STORE.setValue(getStrikethroughPreferenceKey(element),
strikethrough);
    }

    /**
     * Sets whether the specified syntax element is underlined in the preference store
     * @param element The syntax element
     * @param underline Whether the syntax element is underlined
     */
    public static void setUnderline(final SyntaxColoringElement element, final boolean
underline)
    {
        PREFERENCE_STORE.setValue(getUnderlinePreferenceKey(element), underline);
    }

    public static void initializeDefaultPreferences()
    {
        PreferenceConverter.setDefault(PREFERENCE_STORE,
                getColorPreferenceKey(SyntaxColoringElement.DEFAULT_TEXT), new
RGB(0,0,0));
        PreferenceConverter.setDefault(PREFERENCE_STORE,
                getColorPreferenceKey(SyntaxColoringElement.COMMENT), new
RGB(63,127,95));
        PreferenceConverter.setDefault(PREFERENCE_STORE,
                getColorPreferenceKey(SyntaxColoringElement.KEYWORD), new
RGB(127,0,85));
        PREFERENCE_STORE.setDefault(getBoldPreferenceKey(SyntaxColoringElement.KEYWORD)
true);
        PreferenceConverter.setDefault(PREFERENCE_STORE,
                getColorPreferenceKey(SyntaxColoringElement.DECLARATION_KEYWORD), new
RGB(127,0,85));
        PREFERENCE_STORE.setDefault(getBoldPreferenceKey(SyntaxColoringElement.DECLARAT
ION_KEYWORD), true);
        PreferenceConverter.setDefault(PREFERENCE_STORE,
                getColorPreferenceKey(SyntaxColoringElement.STATEMENT_KEYWORD), new
RGB(127,0,85));
        PREFERENCE_STORE.setDefault(getBoldPreferenceKey(SyntaxColoringElement.STATEMEN
T_KEYWORD), true);
        PreferenceConverter.setDefault(PREFERENCE_STORE,
                getColorPreferenceKey(SyntaxColoringElement.VARIABLE), new
RGB(0,0,192));
        PreferenceConverter.setDefault(PREFERENCE_STORE,
                getColorPreferenceKey(SyntaxColoringElement.END_OF_LINE_DOT), new
RGB(255,0,0));
        PreferenceConverter.setDefault(PREFERENCE_STORE,
                getColorPreferenceKey(SyntaxColoringElement.NUMBER), new
RGB(128,128,128));
        PreferenceConverter.setDefault(PREFERENCE_STORE,
                getColorPreferenceKey(SyntaxColoringElement.FUNCTION), new
RGB(130,66,249));
        PreferenceConverter.setDefault(PREFERENCE_STORE,
                getColorPreferenceKey(SyntaxColoringElement.NUMERIC_CONSTANT), new
RGB(0,128,255));
        PreferenceConverter.setDefault(PREFERENCE_STORE,
                getColorPreferenceKey(SyntaxColoringElement.SYMBOLIC_CONSTANT), new
RGB(0,128,255));
        PreferenceConverter.setDefault(PREFERENCE_STORE,
                getColorPreferenceKey(SyntaxColoringElement.IMPLICATION_OPERATOR), new
RGB(127,0,85));
    }

    /**
     * Add a listener for changes in the syntax coloring preferences
```

```
     * @param listener the listener to add
     */
    public static void addSyntaxColoringPreferenceListener(
            final SyntaxColoringPreferenceListener listener)
    {
        if(listener != null && !listeners.contains(listener))
        {
            listeners.add(listener);
        }
    }

    /**
     * Remove a listener for changes in the syntax coloring preferences
     * @param listener the listener to remove
     */
    public static void removeSyntaxColoringPreferenceListener(
            final SyntaxColoringPreferenceListener listener)
    {
        if(listener != null && listeners.contains(listener))
        {
            listeners.remove(listener);
        }
    }

    private static void fireColorUpdated(final SyntaxColoringElement element, final RGB
color)
    {
        for (final Iterator i = listeners.iterator(); i.hasNext();)
        {
            final SyntaxColoringPreferenceListener listener
                = (SyntaxColoringPreferenceListener) i.next();

            listener.colorUpdated(element, color);
        }
    }

    private static void fireBoldUpdated(final SyntaxColoringElement element, final
boolean bold)
    {
        for (final Iterator i = listeners.iterator(); i.hasNext();)
        {
            final SyntaxColoringPreferenceListener listener
                = (SyntaxColoringPreferenceListener) i.next();

            listener.boldUpdated(element, bold);
        }
    }

    private static void fireItalicUpdated(final SyntaxColoringElement element, final
boolean italic)
    {
        for (final Iterator i = listeners.iterator(); i.hasNext();)
        {
            final SyntaxColoringPreferenceListener listener
                = (SyntaxColoringPreferenceListener) i.next();

            listener.italicUpdated(element, italic);
        }
    }

    private static void fireStrikethroughUpdated(final SyntaxColoringElement element,
final boolean strikethrough)
    {
        for (final Iterator i = listeners.iterator(); i.hasNext();)
        {
            final SyntaxColoringPreferenceListener listener
                = (SyntaxColoringPreferenceListener) i.next();

            listener.strikethroughUpdated(element, strikethrough);
        }
    }

    private static void fireUnderlineUpdated(final SyntaxColoringElement element, final
boolean underline)
    {
        for (final Iterator i = listeners.iterator(); i.hasNext();)
        {
            final SyntaxColoringPreferenceListener listener
                = (SyntaxColoringPreferenceListener) i.next();

            listener.underlineUpdated(element, underline);
        }
    }

    //Private methods for creating the preference keys

    private static String getElementPreferenceRoot(final SyntaxColoringElement element)
    {
        return SYNTAX_PREFERENCE_ROOT + "." + element.toString().toLowerCase();
    }

    private static String getColorPreferenceKey(final SyntaxColoringElement element)
    {
        return getElementPreferenceRoot(element) + "." + COLOR_ATTRIBUTE;
    }

    private static String getBoldPreferenceKey(final SyntaxColoringElement element)
    {
        return getElementPreferenceRoot(element) + "." + BOLD_ATTRIBUTE;
    }

    private static String getItalicPreferenceKey(final SyntaxColoringElement element)
    {
        return getElementPreferenceRoot(element) + "." + ITALIC_ATTRIBUTE;
    }

    private static String getUnderlinePreferenceKey(final SyntaxColoringElement
element)
    {
        return getElementPreferenceRoot(element) + "." + UNDERLINE_ATTRIBUTE;
    }

    private static String getStrikethroughPreferenceKey(final SyntaxColoringElement
element)
    {
        return getElementPreferenceRoot(element) + "." + STRIKETHROUGH_ATTRIBUTE;
    }
}
```

```
/*****************************************************************************
 * APE: AnsProlog* Programming Environment plug-in for the Eclipse platform
 * Copyright (C) 2006  Adrian Sureshkumar
 *
 * This program is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License as published by the Free Software
 * Foundation; either version 2 of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT ANY
 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE.  See the GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along with this
 * program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street,
 * Fifth Floor, Boston, MA  02110-1301, USA.
 *****************************************************************************/

package uk.ac.bath.cs.asp.ide.lparse.core.ast;

/**
 * A problem in an lparse source file
 * @author Adrian Sureshkumar
 */
public class LparseSourceProblem
{
    public static final int ERROR = 0;
    public static final int WARNING = 1;
    public static final int INFO = 2;

    private ASTNode node;
    private int type;
    private String description;

    /**
     * Creates a new Lparse source problem
     * @param type The type of problem (ERROR, WARNING, INFO)
     * @param node The node where the problem occurred
     * @param description A description of the problem
     */
    public LparseSourceProblem(final int type,
            final ASTNode node, final String description)
    {
        //Set the type
        if(type < ERROR || type > INFO)
        {
            //Default to info if invalid type specified
            this.type = INFO;
        }
        else
        {
            this.type = type;
        }

        this.node = node;
        this.description = description;
    }

    /**
     * Gets the type of problem (ERROR, WARNING, INFO)
     * @return The typr of problem
     */
    public int getType()
    {
        return type;
    }

    /**
     * Gets the description of the problem
     * @return The problem description
     */
    public String getDescription()
    {
        return description;
    }

    /**
     * Gets the node where the problem occurs
     * @return The node
     */
    public ASTNode getNode()
    {
        return node;
    }
}
```

```
/*******************************************************************************
 * APE: AnsProlog* Programming Environment plug-in for the Eclipse platform
 * Copyright (C) 2006  Adrian Sureshkumar
 *
 * This program is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License as published by the Free Software
 * Foundation; either version 2 of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT ANY
 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE.  See the GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along with this
 * program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street,
 * Fifth Floor, Boston, MA  02110-1301, USA.
 ******************************************************************************/

package uk.ac.bath.cs.asp.ide.lparse.internal.ui.editor;

import java.lang.reflect.InvocationTargetException;

import org.eclipse.core.resources.IMarker;
import org.eclipse.core.resources.IResource;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.jface.text.BadLocationException;
import org.eclipse.jface.text.IDocument;
import org.eclipse.jface.text.IRegion;
import org.eclipse.jface.text.reconciler.DirtyRegion;
import org.eclipse.jface.text.reconciler.IReconcilingStrategy;
import org.eclipse.ui.actions.WorkspaceModifyOperation;

import uk.ac.bath.cs.asp.ide.lparse.core.ast.ASTNode;
import uk.ac.bath.cs.asp.ide.lparse.core.ast.LparseParser;
import uk.ac.bath.cs.asp.ide.lparse.core.ast.LparseSourceFile;
import uk.ac.bath.cs.asp.ide.lparse.core.ast.LparseSourceProblem;
import uk.ac.bath.cs.asp.ide.lparse.ui.LparseUIPlugin;

/**
 * Reconciling strategy for updating the model in the lparse editor
 * @author Adrian Sureshkumar
 */
public class LparseReconcilingStrategy implements IReconcilingStrategy
{
    private IDocument document;
    private LparseSourceProblem[] problems;
    private LparseEditor editor;
    private final LparseParser parser;

    /**
     * Creates a new Lparse reconciling strategy
     * @param editor The editor to be reconciled
     */
    public LparseReconcilingStrategy(final LparseEditor editor)
    {
        this.editor = editor;
        parser = new LparseParser();
    }

    public void setDocument(IDocument document)
    {
        this.document = document;
        reconcile();
    }

    public void reconcile(DirtyRegion dirtyRegion, IRegion subRegion)
    {
        reconcile();
    }

    public void reconcile(IRegion partition)
    {
        reconcile();
    }

    /**
     * Parses the entire document and updates problems and document model
     */
    private void reconcile()
    {
        //Parse the document
        final LparseSourceFile sourceFile = parser.parse(document);

        //Get any problems encountered
        problems = sourceFile.getProblems();

        //Update problem list
        try
        {
            updateProblemMarkers.run(null);
        }
        catch (InvocationTargetException e)
        {
            LparseUIPlugin.getLogger().logException(e);
        }
        catch (InterruptedException e)
        {
            LparseUIPlugin.getLogger().logException(e);
        }

        //Update the UI (from within display thread)
        editor.getSite().getWorkbenchWindow().getWorkbench().getDisplay()
        .asyncExec(new Runnable()
        {
              public void run()
              {
                  //Update the editors model
                  editor.setModel(sourceFile);
              }
        });
    }

    /**
     * Operation to update the problem markers on resources
     */
    private final WorkspaceModifyOperation updateProblemMarkers = new
WorkspaceModifyOperation()
    {
        protected void execute(IProgressMonitor monitor) throws CoreException,
InvocationTargetException, InterruptedException
        {
            final IResource resource =
(IResource)editor.getEditorInput().getAdapter(IResource.class);

            //Delete any existing markers on the resource
            resource.deleteMarkers(IMarker.PROBLEM, true, IResource.DEPTH_INFINITE);

            //Add markers from list of problems
            for (int i = 0; i < problems.length; i++)
            {
                final LparseSourceProblem problem = problems[i];

                //Create a problem marker
                final IMarker marker = resource.createMarker(IMarker.PROBLEM);
                marker.setAttribute(IMarker.PRIORITY,IMarker.PRIORITY_NORMAL);
                marker.setAttribute(IMarker.MESSAGE, problem.getDescription());
```

```
                //Set the severity
                if(problem.getType() == LparseSourceProblem.ERROR)
                {
                    marker.setAttribute(IMarker.SEVERITY,IMarker.SEVERITY_ERROR);
                }
                if(problem.getType() == LparseSourceProblem.WARNING)
                {
                    marker.setAttribute(IMarker.SEVERITY,IMarker.SEVERITY_WARNING);
                }
                if(problem.getType() == LparseSourceProblem.INFO)
                {
                    marker.setAttribute(IMarker.SEVERITY,IMarker.SEVERITY_INFO);
                }

                //Mark problem range
                final ASTNode node = problem.getNode();
                final int offset = node.getStart();
                marker.setAttribute(IMarker.CHAR_START, offset);
                marker.setAttribute(IMarker.CHAR_END, offset + node.getLength());

                //Get the corresponding line
                try
                {
                    marker.setAttribute(IMarker.LINE_NUMBER,
document.getLineOfOffset(offset) + 1);
                }
                catch (BadLocationException e)
                {
                    LparseUIPlugin.getLogger().logException(e);
                }
            }
        }
    };

}
```



```
/*****************************************************************************
 * APE: AnsProlog* Programming Environment plug-in for the Eclipse platform
 * Copyright (C) 2006  Adrian Sureshkumar
 *
 * This program is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License as published by the Free Software
 * Foundation; either version 2 of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT ANY
 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE.  See the GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along with this
 * program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street,
 * Fifth Floor, Boston, MA  02110-1301, USA.
 *****************************************************************************/

package uk.ac.bath.cs.asp.ide.smodels.launch;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.util.LinkedList;
import java.util.List;

import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.Preferences;
import org.eclipse.core.runtime.Status;
import org.eclipse.debug.core.ILaunch;
import org.eclipse.debug.core.ILaunchConfiguration;
import org.eclipse.debug.core.model.ILaunchConfigurationDelegate;
import org.eclipse.debug.core.model.RuntimeProcess;
import uk.ac.bath.cs.asp.ide.lparse.launch.LparseLaunchConfigurationDelegate;
import uk.ac.bath.cs.asp.ide.smodels.ISmodelsPreferenceConstants;
import uk.ac.bath.cs.asp.ide.smodels.SmodelsPlugin;

/**
 * Launch configuration delegate object for launching smodels
 * @author Adrian Sureshkumar
 */
public class SmodelsLaunchConfigurationDelegate implements
        ILaunchConfigurationDelegate
{
    private static final String BACKJUMP_OPTION = "-backjump";
    private static final String NO_LOOK_AHEAD_OPTION = "-nolookahead";
    private static final String SLOPPY_HEURISTIC_OPTION = "-sloppy_heuristic";
    private static final String RANDOMIZE_OPTION = "-randomize";
    private static final String INTERNAL_OPTION = "-internal";
    private static final String WELL_FOUNDED_OPTION = "-w";
    private static final String TRIES_OPTION = "-tries";
    private static final String CONFLICTS_OPTION = "-conflicts";
    private static final String SEED_OPTION = "-seed";

    public void launch(ILaunchConfiguration configuration, String mode,
            ILaunch launch, IProgressMonitor monitor) throws CoreException
    {
        //Build the command line
        final String[] cmdArray = buildCommandLine(configuration);

        //Launch lparse
        final Process lparse = new
 LparseLaunchConfigurationDelegate().launch(configuration);

        if(lparse != null)
        {
```

```
            //Start the process
            Process smodels = null;

            try
            {
                smodels = Runtime.getRuntime().exec(cmdArray);
            }
            catch (IOException e)
            {
                SmodelsPlugin.getLogger().logException(e);
            }

            if(smodels != null)
            {
                //Determine whether to pipe to external process
                Process external = null;
                final boolean pipeToExternal = configuration.getAttribute(
                        ISmodelsLaunchConstants.ATTR_OUTPUT_TO_EXTERNAL_PROGRAM,
false);
                if(pipeToExternal)
                {
                    //Get the command
                    final String externalCommand = configuration.getAttribute(
                            ISmodelsLaunchConstants.ATTR_OUTPUT_COMMAND, "");

                    //Start the process
                    try
                    {
                        external = Runtime.getRuntime().exec(externalCommand);
                    }
                    catch (IOException e)
                    {
                        SmodelsPlugin.getLogger().logException(e);
                    }
                }

                //Place in a runtime process
                new RuntimeProcess(launch, (external != null) ? external : smodels,
"Smodels", null);

                //Pipe lparse output to smodels
                pipe(lparse, smodels);

                if(external != null)
                {
                    //Pipe smodels output to external program
                    pipe(smodels, external);
                }
            }
            else
            {
                //Kill the lparse process
                lparse.destroy();
            }
        }
    }

    /**
     * Builds the command line from the configuration
     * @param The configuration
     * @return The command line array
     * @throws CoreException
     */
    private String[] buildCommandLine(final ILaunchConfiguration configuration) throws
CoreException
    {
        final List command = new LinkedList();

        //Get the program path
        final Preferences preferences =
 SmodelsPlugin.getDefault().getPluginPreferences();
        final String program =
 preferences.getString(ISmodelsPreferenceConstants.SMODELS_PROGRAM_LOCATION);

        if(program.equals(""))
        {
            throw new CoreException(new Status(Status.ERROR,
                    SmodelsPlugin.getDefault().getBundle().getSymbolicName(),
                    Status.OK,
                    "Smodels could not be launched as the program location has not been
configured.\n\n"
                    + "Please set the smodels location in the ASP preferences dialog
(Window->Preferences).",
                    null));
        }

        command.add(program);

        //Load argument values

        //Number of models to compute
        final int noOfModels = configuration.getAttribute(
                ISmodelsLaunchConstants.ATTR_NO_OF_MODELS, 0);

        command.add("" + noOfModels);

        //Backjump option
        final boolean backjump = configuration.getAttribute(
                ISmodelsLaunchConstants.ATTR_BACKJUMP, false);
        if(backjump)
        {
            command.add(BACKJUMP_OPTION);
        }

        //No look ahead option
        final boolean noLookAhead = configuration.getAttribute(
                ISmodelsLaunchConstants.ATTR_NO_LOOK_AHEAD, false);
        if(noLookAhead)
        {
            command.add(NO_LOOK_AHEAD_OPTION);
        }

        //Sloppy heristics option
        final boolean sloppyHeursitics = configuration.getAttribute(
                ISmodelsLaunchConstants.ATTR_SLOPPY_HEURSITIC, false);
        if(sloppyHeursitics)
        {
            command.add(SLOPPY_HEURISTIC_OPTION);
        }

        //Randomize option
        final boolean randomize = configuration.getAttribute(
                ISmodelsLaunchConstants.ATTR_RANDOMIZE, false);
        if(randomize)
        {
            command.add(RANDOMIZE_OPTION);
        }

        //internal option
        final boolean internal = configuration.getAttribute(
                ISmodelsLaunchConstants.ATTR_INTERNAL, false);
        if(internal)
        {
            command.add(INTERNAL_OPTION);
        }
```

```
        //Well founded model option
        final boolean wellFounded = configuration.getAttribute(
                ISmodelsLaunchConstants.ATTR_WELL_FOUNDED, false);
        if(wellFounded)
        {
            command.add(WELL_FOUNDED_OPTION);
        }

        //Number of tries option
        final boolean useNoOfTries = configuration.getAttribute(
                ISmodelsLaunchConstants.ATTR_USE_NO_OF_TRIES, false);
        if(useNoOfTries)
        {
            final int noOfTries = configuration.getAttribute(
                    ISmodelsLaunchConstants.ATTR_NO_OF_TRIES, 0);
            command.add(TRIES_OPTION);
            command.add("" + noOfTries);
        }

        //Number of conflicts option
        final boolean useNoOfConflicts = configuration.getAttribute(
                ISmodelsLaunchConstants.ATTR_USE_NO_OF_CONFLICTS, false);
        if(useNoOfConflicts)
        {
            final int noOfConflicts = configuration.getAttribute(
                    ISmodelsLaunchConstants.ATTR_NO_OF_CONFLICTS, 0);
            command.add(CONFLICTS_OPTION);
            command.add("" + noOfConflicts);
        }

        //Seed option
        final boolean useSeed = configuration.getAttribute(
                ISmodelsLaunchConstants.ATTR_USE_SEED, false);
        if(useSeed)
        {
            final int seed = configuration.getAttribute(
                    ISmodelsLaunchConstants.ATTR_SEED, 0);
            command.add(SEED_OPTION);
            command.add("" + seed);
        }

        return (String[])command.toArray(new String[command.size()]);
    }

    /**
     * Pipes data from one process to another
     * @param source The source process
     * @param sink The sink process
     * @throws CoreException
     */
    private void pipe(final Process source, final Process sink) throws CoreException
    {
        //Get buffered readers and writers for the process streams
        final BufferedReader reader =
            new BufferedReader(new InputStreamReader(source.getInputStream()));
        final BufferedWriter writer =
            new BufferedWriter(new OutputStreamWriter(sink.getOutputStream()));

        //Copy data from one stream into the other
        String line = null;
        try
        {
            while((line = reader.readLine()) != null)
            {
                writer.write(line);
                writer.newLine();
            }
            writer.flush();
        }
        catch (IOException e)
        {
            SmodelsPlugin.getLogger().logException(e);
            throw new CoreException(new Status(Status.ERROR,
                    SmodelsPlugin.getDefault().getBundle().getSymbolicName(),
                    Status.OK,
                    "Error launching smodels: IO error whilst piping",
                    e));
        }
        finally
        {
            //Close the streams
            try
            {
                reader.close();
            }
            catch (IOException e) {}

            try
            {
                writer.close();
            }
            catch (IOException e) {}
        }
    }
}
```

```
/*****************************************************************************
 * APE: AnsProlog* Programming Environment plug-in for the Eclipse platform
 * Copyright (C) 2006  Adrian Sureshkumar
 *
 * This program is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License as published by the Free Software
 * Foundation; either version 2 of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT ANY
 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE.  See the GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along with this
 * program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street,
 * Fifth Floor, Boston, MA  02110-1301, USA.
 *****************************************************************************/

package uk.ac.bath.cs.asp.ide.lparse.core.ast;

import java.util.LinkedList;
import java.util.List;

/**
 * A model of an Lparse Source file
 * @author Adrian Sureshkumar
 */
public class LparseSourceFile extends NonTerminalASTNode
{
    private List problems;
    private SymbolTable symbolicConstantTable;
    private SymbolTable numericConstantTable;
    private SymbolTable variableTable;
    private SymbolTable functionTable;
    private SymbolTable symbolicFunctionTable;
    private SymbolTable atomTable;

    /**
     * Creates a new LparseSourceFile node
     * @param length
     */
    LparseSourceFile()
    {
        problems = new LinkedList();
        symbolicConstantTable = new SymbolTable();
        numericConstantTable = new SymbolTable();
        variableTable = new SymbolTable();
        functionTable = new SymbolTable();
        symbolicFunctionTable = new SymbolTable();
        atomTable = new SymbolTable();
    }

    /**
     * Associate problems with the source file
     * @param problems
     */
    void addProblem(final LparseSourceProblem problem)
    {
        problems.add(problem);
    }

    /**
     * Gets the comments in the source file
     * @return The comments in the source file
     */
    public Comment[] getComments()
    {
        final List comments = getChildren(Comment.class);
        return (Comment[])comments.toArray(new Comment[comments.size()]);
    }

    /**
     * Gets the rules in the source file
     * @return The rules in the source file
     */
    public Rule[] getRules()
    {
        final List rules = getChildren(Rule.class);
        return (Rule[])rules.toArray(new Rule[rules.size()]);
    }

    /**
     * Gets the problems associtaed with this source file
     * @return the list of problems
     */
    public LparseSourceProblem[] getProblems()
    {
        return (LparseSourceProblem[])problems.toArray(new
LparseSourceProblem[problems.size()]);
    }

    /**
     * Get the function symbol table
     * @return The function symbol table
     */
    public SymbolTable getFunctionTable()
    {
        return functionTable;
    }

    /**
     * Get the symbolic function symbol table
     * @return The symbolic function symbol table
     */
    public SymbolTable getSymbolicFunctionTable()
    {
        return symbolicFunctionTable;
    }

    /**
     * Get the numeric constant symbol table
     * @return The numeric constant symbol table
     */
    public SymbolTable getNumericConstantTable()
    {
        return numericConstantTable;
    }

    /**
     * Get the symbolic constant symbol table
     * @return The symbolic constant symbol table
     */
    public SymbolTable getSymbolicConstantTable()
    {
        return symbolicConstantTable;
    }

    /**
     * Get the variable symbol table
     * @return The variable symbol table
     */
    public SymbolTable getVariableTable()
    {
        return variableTable;
    }

    /**
```

```
     * Get the atom symbol table
     * @return The atom symbol table
     */
    public SymbolTable getAtomTable()
    {
        return atomTable;
    }
}
```



```
/*****************************************************************************
 * APE: AnsProlog* Programming Environment plug-in for the Eclipse platform
 * Copyright (C) 2006  Adrian Sureshkumar
 *
 * This program is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License as published by the Free Software
 * Foundation; either version 2 of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT ANY
 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE.  See the GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along with this
 * program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street,
 * Fifth Floor, Boston, MA  02110-1301, USA.
 *****************************************************************************/

package uk.ac.bath.cs.asp.ide.lparse.internal.ui.editor.actions;

import org.eclipse.jface.action.Action;
import org.eclipse.jface.text.BadLocationException;
import org.eclipse.jface.text.IDocument;
import org.eclipse.jface.text.TextSelection;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.text.edits.DeleteEdit;
import org.eclipse.text.edits.InsertEdit;
import org.eclipse.text.edits.MultiTextEdit;
import org.eclipse.ui.texteditor.ITextEditor;

import uk.ac.bath.cs.asp.ide.lparse.ui.LparseUIPlugin;

/**
 * Action to toggle comments in the lparse editor
 * @author Adrian Sureshkumar
 */
public class ToggleCommentAction extends Action
{
    ITextEditor editor;

    public ToggleCommentAction()
    {
        super("Toggle Comment");
        setEnabled(false);
    }

    /**
     * Set the editor to toggle comments for
     * @param editor The editor
     */
    public void setEditor(final ITextEditor editor)
    {
        this.editor = editor;
        setEnabled(editor != null);
    }

    public void run()
    {
        final ISelection selection = editor.getSelectionProvider().getSelection();

        if(selection instanceof TextSelection)
        {
            final TextSelection textSelection = (TextSelection)selection;
            final IDocument document =
editor.getDocumentProvider().getDocument(editor.getEditorInput());

            int start = textSelection.getStartLine();
            int end = textSelection.getEndLine();
```

```
            //Create edits for commenting and uncommenting
            final MultiTextEdit commentEdit = new MultiTextEdit();
            final MultiTextEdit uncommentEdit = new MultiTextEdit();

            boolean uncomment = true;

            try
            {
                for(int l = start; l <= end; l++)
                {
                    //Get line of text
                    final int lineStart = document.getLineOffset(l);
                    final int lineLength = document.getLineLength(l);
                    final String line = document.get(lineStart, lineLength);

                    //Determine if its already commented
                    if(line.trim().startsWith("%"))
                    {
                        //Delete the comment character
                        int commentPos = lineStart + line.indexOf("%");
                        uncommentEdit.addChild(new DeleteEdit(commentPos, 1));
                    }
                    else
                    {
                        uncomment = false;
                    }

                    //Comment the line
                    commentEdit.addChild(new InsertEdit(lineStart, "%"));
                }

                //Apply the edits
                if(uncomment)
                {
                    uncommentEdit.apply(document);
                }
                else
                {
                    commentEdit.apply(document);
                }

                //Reselect the text - if it was previously selected
                final int selStart = textSelection.getOffset();
                final int selLength = textSelection.getLength();

                if(selLength > 0)
                {
                    //Calculate the change in length caused by the edit
                    int editLength = 1 + end - start;
                    if(uncomment)
                    {
                        editLength = -editLength;
                    }

                    editor.getSelectionProvider().setSelection(
                            new TextSelection(selStart, selLength + editLength));
                }
            }
            catch (BadLocationException e)
            {
                LparseUIPlugin.getLogger().logException(e);
            }
        }
    }
}
```



```
/*****************************************************************************
 * APE: AnsProlog* Programming Environment plug-in for the Eclipse platform
 * Copyright (C) 2006  Adrian Sureshkumar
 *
 * This program is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License as published by the Free Software
 * Foundation; either version 2 of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT ANY
 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE.  See the GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along with this
 * program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street,
 * Fifth Floor, Boston, MA  02110-1301, USA.
 *****************************************************************************/

package uk.ac.bath.cs.asp.ide.lparse.internal.dependencygraph;

import java.util.LinkedList;
import java.util.List;

import uk.ac.bath.cs.asp.ide.dependencygraphs.model.AnsPrologProgram;
import uk.ac.bath.cs.asp.ide.dependencygraphs.model.Literal;
import uk.ac.bath.cs.asp.ide.dependencygraphs.model.Rule;
import uk.ac.bath.cs.asp.ide.lparse.core.ast.Atom;
import uk.ac.bath.cs.asp.ide.lparse.core.ast.Head;
import uk.ac.bath.cs.asp.ide.lparse.core.ast.HeadList;
import uk.ac.bath.cs.asp.ide.lparse.core.ast.LiteralList;
import uk.ac.bath.cs.asp.ide.lparse.core.ast.LparseSourceFile;
import uk.ac.bath.cs.asp.ide.lparse.core.ast.OrderedDisjunction;
import uk.ac.bath.cs.asp.ide.lparse.core.ast.Tail;

/**
 * Utility class to transform an Lparse source file model into a an AnsProlog program
 model for
 * use in dependency graphs
 * @author Adrian Sureshkumar
 */
public class DependencyGraphBuilder
{
    //Prevent instantiation
    private DependencyGraphBuilder() {}

    /**
     * Creates a model of an AnsProlog* program for use in a dependency graph
     * @param sourceFile The source file to create the model for
     * @return The model of the program
     */
    public static AnsPrologProgram createDependencyGraphModel(final LparseSourceFile
 sourceFile)
    {
        final AnsPrologProgram program = new AnsPrologProgram();

        if(sourceFile != null)
        {
            //Add rules to the program
            final uk.ac.bath.cs.asp.ide.lparse.core.ast.Rule[] rules =
 sourceFile.getRules();
            for (int i = 0; i < rules.length; i++)
            {
                final Rule rule = new Rule();

                //Add head atoms
                final Head head = rules[i].getHead();
                if(head != null)
                {
                    final Atom[] atoms = getAtoms(head);
```

```
                    for (int j = 0; j < atoms.length; j++)
                    {
                        final String predicateSymbol =
atoms[j].getPredicateSymbol().getText();
                        rule.addLiteralToHead(new Literal(predicateSymbol));
                    }
                }

                //Add body literals
                final Tail tail = rules[i].getTail();
                if(tail != null)
                {
                    final uk.ac.bath.cs.asp.ide.lparse.core.ast.Literal[] literals =
getLiterals(tail);
                    for (int j = 0; j < literals.length; j++)
                    {
                        final String predicateSymbol =
literals[j].getAtom().getPredicateSymbol().getText();
                        final boolean positive = !literals[j].isNegated();
                        rule.addLiteralToBody(new Literal(predicateSymbol, positive));
                    }
                }

                program.addRule(rule);
            }
        }

        return program;
    }

    private static Atom[] getAtoms(final Head head)
    {
        //Build the atom list
        final List atoms = new LinkedList();

        //Get any atoms in the head
        List children = head.getChildren(Atom.class);
        if(children.size() > 0)
        {
            atoms.addAll(children);
        }

        //Get from headlist
        children = head.getChildren(HeadList.class);
        if(children.size() > 0)
        {
            final HeadList headList = (HeadList)children.get(0);
            atoms.addAll(headList.getChildren(Atom.class));
        }

        //Get from ordered disjunction
        children = head.getChildren(OrderedDisjunction.class);
        if(children.size() > 0)
        {
            final OrderedDisjunction disjunction = (OrderedDisjunction)children.get(0);
            atoms.addAll(disjunction.getChildren(Atom.class));
        }

        //Get from literal list
        children = head.getChildren(LiteralList.class);
        if(children.size() > 0)
        {
            final LiteralList literalList = (LiteralList)children.get(0);
            children =
literalList.getChildren(uk.ac.bath.cs.asp.ide.lparse.core.ast.Literal.class);
            for(int i = 0; i < children.size(); i++)
            {
                //Add atom from literal
                final uk.ac.bath.cs.asp.ide.lparse.core.ast.Literal literal =
                    (uk.ac.bath.cs.asp.ide.lparse.core.ast.Literal)children.get(i);;
                atoms.add(literal.getAtom());
            }
        }

        return (Atom[])atoms.toArray(new Atom[atoms.size()]);
    }

    private static uk.ac.bath.cs.asp.ide.lparse.core.ast.Literal[] getLiterals(final
 Tail tail)
    {
        //Build the literal list
        final List literals = new LinkedList();

        literals.addAll(tail.getChildren(uk.ac.bath.cs.asp.ide.lparse.core.ast.Literal.c
lass));

        //Deal with special tails
        final List specialTails = tail.getChildren(Tail.class);
        if(specialTails.size() > 0)
        {
            for(int i = 0; i < specialTails.size(); i++)
            {
                final Tail specialTail = (Tail)specialTails.get(i);
                literals.addAll(specialTail.getChildren(uk.ac.bath.cs.asp.ide.lparse.co
re.ast.Literal.class));
            }
        }

        return (uk.ac.bath.cs.asp.ide.lparse.core.ast.Literal[])literals.toArray(
                new uk.ac.bath.cs.asp.ide.lparse.core.ast.Literal[literals.size()]);
    }
}
```

```
/*****************************************************************************
 * APE: AnsProlog* Programming Environment plug-in for the Eclipse platform
 * Copyright (C) 2006  Adrian Sureshkumar
 *
 * This program is free software; you can redistribute it and/or modify it under the
 * terms of the GNU General Public License as published by the Free Software
 * Foundation; either version 2 of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT ANY
 * WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE.  See the GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along with this
 * program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street,
 * Fifth Floor, Boston, MA  02110-1301, USA.
 *****************************************************************************/

package uk.ac.bath.cs.asp.ide.dependencygraphs.ui.views;

import java.util.HashMap;
import java.util.Map;

import org.eclipse.draw2d.FigureUtilities;
import org.eclipse.draw2d.graph.DirectedGraph;
import org.eclipse.draw2d.graph.DirectedGraphLayout;
import org.eclipse.draw2d.graph.Edge;
import org.eclipse.draw2d.graph.EdgeList;
import org.eclipse.draw2d.graph.Node;
import org.eclipse.draw2d.graph.NodeList;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.part.ViewPart;

import uk.ac.bath.cs.asp.ide.dependencygraphs.model.AnsPrologProgram;
import uk.ac.bath.cs.asp.ide.dependencygraphs.model.Literal;
import uk.ac.bath.cs.asp.ide.dependencygraphs.model.Rule;
import uk.ac.bath.cs.asp.ide.dependencygraphs.ui.viewers.GraphViewer;

/**
 * A view for displaying dependency graphs
 * @author Adrian Sureshkumar
 */
public class DependencyGraphView extends ViewPart
{
    public static final String VIEW_ID =
 "uk.ac.bath.cs.asp.ide.dependencygraphs.ui.views.DependencyGraphView";

    private GraphViewer viewer;

    public void createPartControl(Composite parent)
    {
        //Set the layout
        parent.setLayout(new FillLayout());

        //Add the graph viewer
        viewer = new GraphViewer(parent);
    }

    public void setFocus()
    {
        // TODO Auto-generated method stub

    }

    /**
     * Display the dependency graph for the given AnsProlog* program in the view
     * @param program The program to display the graph for
     */
    public void displayDependencyGraph(final AnsPrologProgram program)
    {
        //If the program is null just clear the viewer
        if(program == null)
        {
            viewer.clear();
            return;
        }

        //Create a manager for the graph
        final GraphManager graphManager = new GraphManager();

        //Add the rules to the graph
        final Rule[] rules = program.getRules();
        for (int r = 0; r < rules.length; r++)
        {
            final Literal[] head = rules[r].getHeadLiterals();
            final Literal[] tail = rules[r].getBodyLiterals();

            //Only add nodes if the head and tail are non-empty
            if(head.length > 0 && tail.length > 0)
            {
                for (int h = 0; h < head.length; h++)
                {
                    final Node headNode =
graphManager.addNode(head[h].getPredicateSymbol());

                    for (int t = 0; t < tail.length; t++)
                    {
                        final Literal tailLit = tail[t];
                        final Node tailNode =
graphManager.addNode(tailLit.getPredicateSymbol());

                        if(tailNode != headNode)
                        {
                            //Create an edge from tail to head
                            graphManager.addEdge(tailNode, headNode,
tailLit.isPositive());
                        }
                        else
                        {
                            //TODO handle nodes dependent on themselves
                        }
                    }

                }
            }
        }

        //Create a graph object
        final DirectedGraph graph = graphManager.getGraph();

        //Lay the graph out
        if(graph.nodes.size() > 0)
        {
            layoutGraph(graph);
        }

        //Display the graph in the viewer
        viewer.displayGraph(graph);
    }

    /**
     * Helper class to manage nodes and sourceToTargetNode in the graph
     */
    private final class GraphManager
    {
        private static final String POSITIVE_LABEL = " + ";
```

```
        private static final String NEGATIVE_LABEL = " - ";
        private static final String BOTH_LABEL = " +- ";

        private final DirectedGraph graph;

        //Indexed structure for the nodes and edges
        private final Map nodes;
        private final Map sourceToTargetNode;

        public GraphManager()
        {
            graph = new DirectedGraph();

            nodes = new HashMap();
            sourceToTargetNode = new HashMap();
        }

        /**
         * Adds the node associated with a predicate symbol to the manager
         * @param predicateSymbol The predicate symbol
         * @return The added node
         */
        private Node addNode(final String predicateSymbol)
        {
            final Node node;

            if(nodes.containsKey(predicateSymbol))
            {
                node = (Node)nodes.get(predicateSymbol);
            }
            else
            {
                node = new Node();
                node.data = predicateSymbol;

                //Ensure that the node is wide enough to contain the text
                final int requiredWidth  =
FigureUtilities.getTextWidth(predicateSymbol,
                        viewer.getControl().getFont()) + 16;

                if(requiredWidth > node.width)
                {
                    node.width = requiredWidth;
                }

                //Add the node to the map
                nodes.put(predicateSymbol, node);

                //Add the node to the node list
                graph.nodes.add(node);
            }

            return node;
        }

        /**
         * Adds the edge going from the source to target node to the manager
         * @param source The source node
         * @param target The target node
         * @param positive Whether the edge is labelled positively or negatively
         * @return The added edge
         */
        public Edge addEdge(final Node source, final Node target, final boolean
positive)
        {
            //Get the map relating target nodes to edges for the source node
            final Map targetNodeToEdge;
            if(!sourceToTargetNode.containsKey(source))
            {
                //Create a map of target nodes from the source node
                targetNodeToEdge = new HashMap();
                sourceToTargetNode.put(source, targetNodeToEdge);
            }
            else
            {
                targetNodeToEdge = (Map)sourceToTargetNode.get(source);
            }

            //Get the edge
            final Edge edge;

            //If the source has the target node - get the edge between them
            if(targetNodeToEdge.containsKey(target))
            {
                edge = (Edge)targetNodeToEdge.get(target);
            }
            else
            {
                edge = new Edge(source, target);

                //Add the edge to the map
                targetNodeToEdge.put(target, edge);

                //Add the edge to the graph
                graph.edges.add(edge);
            }

            //Label the edge
            labelEdge(edge, positive);

            return edge;
        }

        /**
         * Updates the label for the given edge
         * @param edge The edge to label
         * @param positive Whether it is labelled as positive
         */
        private void labelEdge(final Edge edge, final boolean positive)
        {
            //Label the edge
            if(edge.data == null)
            {
                edge.data = (positive) ? POSITIVE_LABEL : NEGATIVE_LABEL;
            }
            else if((edge.data == POSITIVE_LABEL && !positive) ||
                    (edge.data == NEGATIVE_LABEL && positive))
            {
                edge.data = BOTH_LABEL;
            }
        }

        /**
         * Gets the graph controlled by the manager
         * @return The graph
         */
        public DirectedGraph getGraph()
        {
            return graph;
        }
    }

    /**
     * Lays out the specified Directed Graph
     * @param graph The graph to lay out
     */
```

```
    private void layoutGraph(final DirectedGraph graph)
    {
        //Check the graph is connected - if not add dummy sourceToTargetNode
        EdgeList dummyEdges = null;

        //Only need to connect graph if more than one node
        if(graph.nodes.size() > 1)
        {
            //Get a copy of the graph's node list
            final NodeList graphNodes = new NodeList();
            graphNodes.addAll(graph.nodes);

            //A list of nodes currently in the tree
            final NodeList treeNodes = new NodeList();

            //Initialise list of nodes with first node in the list
            treeNodes.add(graphNodes.get(0));
            graphNodes.remove(graphNodes.get(0));

            //Get a copy of the graph's edge list
            final EdgeList graphEdges = new EdgeList();
            graphEdges.addAll(graph.edges);

            //Create a list to store any dummy sourceToTargetNode created
            dummyEdges = new EdgeList();

            //Add all the nodes to the tree
            while(!graphNodes.isEmpty())
            {
                //Find a node connected to the current tree by an edge
                Node node = null;
                for(int i = 0; node == null && i < graphEdges.size(); i++)
                {
                    final Edge edge = graphEdges.getEdge(i);

                    //If the edge connects two nodes in the tree remove it from the
list
                    if(treeNodes.contains(edge.source) &&
 treeNodes.contains(edge.target))
                    {
                        graphEdges.remove(edge);
                    }
                    //Check whether the edge connects a node to the tree
                    else if(treeNodes.contains(edge.source))
                    {
                        node = edge.target;
                        graphEdges.remove(edge);
                    }
                    else if(treeNodes.contains(edge.target))
                    {
                        node = edge.source;
                        graphEdges.remove(edge);
                    }
                }

                //If no node found
                if(node == null)
                {
                    //Choose the next node in the list
                    node = graphNodes.getNode(0);

                    //Connect it to the last node in the list with a dummy edge
                    dummyEdges.add(new Edge(node, treeNodes.getNode(treeNodes.size() -
1)));
                }

                //Add the node to the list of nodes in the tree
                treeNodes.add(node);
                graphNodes.remove(node);
            }

            //Add the dummy sourceToTargetNode to the list
            graph.edges.addAll(dummyEdges);
        }

        //Lay out the graph
        final DirectedGraphLayout graphLayout = new DirectedGraphLayout();
        graphLayout.visit(graph);

        //Remove the dummy sourceToTargetNode
        if(dummyEdges != null )
        {
            graph.edges.removeAll(dummyEdges);
        }
    }
}
```